TU Clausthal

Clausthal University of Technology

Software-Orientierter Programmierstil bei VHDL

Harald Richter

IfI Technical Report Series **IfI-10-07**

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Michael Köster
Contact: michael.koester@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Software-Orientierter

Programmierstil bei VHDL

Prof. Dr. Harald Richter

Technische Universität Clausthal

Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany

E-Mail: richter@in.tu-clausthal.de

Inhaltsverzeichnis

1 Abstract

Der hier beschriebene Software-orientierte Programmierstil für VHDL stellt eine innovative Neuentwicklung auf dem Gebiet der Synthese programmierbarer Logikschaltkreise (FPGAs) dar. Das Ziel des Software-orientierter Stils ist die einfache Erstellung synthesefähiger Algorithmen in VHDL. Es wird anhand von Beispielprogrammen und mit Hilfe der Synthesewerkzeuge xst von Xilinx und Synplify von Synopsys, sowie des VHDL-Simulators Modelsim von Mentor Graphics gezeigt, wie dieses Ziel erreicht werden kann. Mit Hilfe des Software-orientierten Programmierstil lassen sich beliebige (mathematische) Algorithmen und Rechnernetzprotokolle in VHDL programmieren, die sicher synthetisiert, in das FPGA geladen und anschließend vom Logik-Chip in Echtzeit ausgeführt werden können. Der neue Programmierstil basiert auf einer Teilnenge des VHDL-Sprachumfangs und einem Satz von Regeln, die bei der Kodierung anzuwenden sind. Simulationen spielen nur als Testhilfe eine Rolle. Mit dem Programmierstil kann die wachsende Kluft zwischen dem grossen Potential, das moderne FPGAs haben, und der limitierten Software, die diese Chips programmieren, effizient und effektiv verkleinert werden.

2 Einleitung

Der Anlass für die Entwicklung eines neuen VHDL-Programmierstils war der Wunsch, über ein lokales Netzwerk (LAN) mit Protokollen zu verfügen, das Bandbreiten oberhalb von 1 Gbit/s und Latenzen unterhalb von 1µs in deterministischer Weise bereitstellen kann. Als weitere Randbedingungen kamen hinzu, dass der Energie- und Volumenbedarf für das gewünschte Echtzeit-LAN und seine Protokolle möglichst klein sein sollte, da beides in einem Automobil einzusetzen war, wo nur begrenzter Platz und eine Energieversorgung, die bereits am Limit ist, zur Verfügung stehen. Schliesslich sollten die Protokolle möglichst alle Funktionen des ISO-7-Schichten-Modell implementieren. um den Anwendungen, d.h. den Steuergeräten im Fahrzeug, den sog. Electronic Controller Units (ECUs) eine mächtige und komfortable Programmierschnittstelle bieten zu können.
Eine Analyse existierender Protokolle in Rechnernetzen und in der Automatisierungstechnik ergab, dass ein solches LAN nicht existierte. Das lieferte den Startschuss für das CarRing II-Projekt [1][2][3], in dem Schritt für Schritt ein Echtzeit-LAN mit Protokollen auf allen 7 ISO-Schichten geschaffen wird.

3 Stand der Technik

3.1 Fortschritte bei FPGAs

Programmierbare Logikbausteine aus der Kategorie der Field Programmable Gate Arrays (FPGAs) haben in den vergangenen Jahren erhebliche Steigerungen in der Anzahl und in der Komplexität der darin enthaltenen Logikzellen erfahren, wodurch die Leistungsfähigkeit und die vielseitige Einsetzbarkeit von FPGAs dramatisch erhöht worden ist. Ein Virtex 7 Chip von

der Fa. Xilinx, beispielsweise, ist in der Lage ca. 400 Prozessoren der 32 Bit-Klasse gleichzeitig auf demselben FPGA-Chip zu emulieren.

3.2 Stagnation bei CPUs

Bemerkenswerterweise kam es in derselben Zeit, in der die FPGA-Performanz förmlich explodierte kaum mehr zu substantiellen Steigerungen der Rechenleistung bei Prozessoren. Dafür sind sowohl thermische Gründe verantwortlich, als auch die sehr hohe inhärente Komplexität in heutigen Prozessorarchitekturen, die bei vertretbarem Testaufwand kaum mehr erhöht werden kann, soll das unternehmerische Risiko der Herstellerfirma kalkulierbar bleibten Dennoch kostete die Entwicklung eines neuen PCs-Prozessers die Fa. Intel bereits in der Vergangenheit mehrere Milliarden Dollar, eine Investition, die sich nur noch sehr wenige Prozessorhersteller leisten konnten.
Neben kommerziellen und architektonischen Gründen begrenzen aber auch thermische Probleme weitere Steigerungen der Rechenleistung eines einzelnen Prozessors, denn bei mehr als ca. 3-4 GHz an Taktrate auf dem Chip wird eine teure Flüssigkeitskühlung zur Wärmeabfuhr erforderlich, was für den Massenmarkt bestehend aus PCs, Laptops, Handhelds, MP3 Playern und Mobiltelefonen nicht darstellbar ist. Bei Prozessoren geht deshalb der Trend in Richtung Reduzierung des Stromverbrauchs und in Richtung Reduzierung der Komplexität der Architektur, mit dem Resultat, dass bei gleichen Kosten mehr als ein Prozessor auf demselben Chip integriert werden kann. Daraus sind die sog. Multi Core und Many Core CPUs entstanden, die bei PCs und Laptops mittlerweile standardmäßig eingesetzt werden.

3.3 Probleme bei der Multicore-Programmierung

Das Problem der stagnierenden Rechenleistung wird durch Verdopplung (Dual Core) oder Vervielfachung der rechnenden Kerne pro Chip nicht gelöst, sondern nur auf die Software verlagert, denn Multi- und Many Core CPUs erfordern eine andere Art der Programmierung, die sog. Parallelrechner-Programmierung. Obwohl das Know How zur Parallelrechner-Programmierung seit vielen Jahren in der Welt der Großrechner existiert und zahlreiche Werkzeuge zur Unterstützung der parallelen Programmierung geschaffen wurden, ist diese immer noch rel. wenigen Spezialisten vorbehalten, die auf Supercomputern in Rechenzentren oder auf PC-Clustern von Firmen wie Google oder Amazon große Anwendungen laufen lassen und dafür mit hohem Zeitaufwand Spezial-Software entwickeln. Parallelisierte Standardanwendungen für PCs existieren hingegen kaum, mit der Folge, dass man Mehrkern-PCs zwar kaufen, aber nicht effizient nutzen kann, da die darauf lauffähigen parallelen Anwendungen fehlen. Für reine Office-Programme wie Word, Power Point oder Excel oder für web-Browser wird es darüberhinaus parallelisierte Versionen auch nie geben, da dort Geschwindigkeit nicht das Problem ist. Dadurch wird die Lücke zwischen der Hardware einerseits und der Software andererseits, d.h. zwischen dem, was der Rechner könnte und zwischen dem, was die Programme leisten und deren Anwender benötigen, immer grösser, da Office-Applikationen bei weitem die Mehrheit aller auf PCs ausgeführten Programme darstellen.

3.4 Algorithmen in Hardware

In dieser Situation bekommen die enormen Leistungssteigerungen von FPGAs eine zusätzliche Bedeutung, denn man kann erwarten, dass in Zukunft bestimmte Anwendungen vermehrt in Hardware auf einem FPGA implementiert werden. Die hochspezialisierten Graphikprozessoren (GPUs) haben bereits erfolgreich demonstriert, dass es für bestimmte Problemklassen, nämlich den sog. Single Instruction Multiple Data-Anwendungen (SIMD) möglich ist, klassische Vielzweck-Prozessoren in PCs hinsichtlich der Rechenleistung weit zu übertreffen (bei gleichen oder geringeren Kosten). Der nächste logische Schritt besteht deshalb darin, die Software direkt in Hardware umzusetzen. So ist es denkbar, dass in Zukunft z.B. Datenbanken wie Oracle oder MySQL auf FPGAs laufen, um so eine fast beliebige Hardware-Unterstützung und -Beschleunigung der dort zu verarbeiteten Datenstrukturen, den sog. B-Trees bzw. Derivate davon, zu erzielen.

3.5 FPGAs für eingebettete Systeme

Ein anderer Trend beim Stand der Technik ist ebenfalls klar erkennbar, und zwar, dass auf dem Gebiet der eingebetteten Systeme, das über Jahrzehnte die klassische Domäne der Mikrocontroller war, zunehmend FPGAs eingesetzt werden. Der Grund dafür ist die begrenzte Leistungsfähigkeit von Mikrocontrollern, die in Echtzeit erbracht werden kann. Speziell bei klar umgrenzten Aufgabenstellungen sind FPGAs gegenüber Mikrocontrollern geschwindigkeitsmäßig im Vorteil, da alles in Hardware implementiert werden kann. Die Voraussetzung ist, dass es gelingt, den gewünschten Algorithmus in einer Hardware-Beschreibungssprache wie VHDL (VLSI Hardware Description Language) [4] oder Verilog darzustellen und daraus eine sog. Netzliste für das FPGA zu synthetisieren. Für sehr schnell auszuführende und stets wiederkehrende, einfachere Aufgaben lohnt es sich bereits heute, ein FPGA anstelle eines Mikrocontrollers in VHDL zu programmieren.

3.6 Software für FPGAs

Bei FPGAs finden derzeit zwei weitere Entwicklungen statt, die für die Diskussion des Standes der Technik wichtig sind: zum einen werden die Programme für FPGAs, die in VHDL oder Verilog verfasst sind, immer komplexer und aufwendiger, da die Chips immer größer werden. Zum anderen können derzeitige FPGAs bereits mehrere Dutzend klassischer Prozessoren gleichzeitig nachahmen, bei allerdings noch mäßigen Taktraten im Bereich von 50-200 MHz. In Zukunft werden Hunderte von 32-Bit-Prozessoren von einem einzigen FPGA-Chip emuliert werden können. Solche Prozessoremulationen sind nichts anderes als VHDL- oder Verilog-Programme, die die Funktion eines klassischen Mikroprozessors beschreiben. Sie heißen deshalb Soft-Core-Prozessoren und erlauben es, dass Anwendungen für eingebettete Systeme in C, C++ oder Java programmiert werden können, denn für Soft-Core-Prozessoren gibt es solche Hochsprachen-Compiler, z.B. im Rahmen des GNU-Projekts. Die Hochsprachenprogrammierung von FPGAs über den Umweg darauf emulierter Standardprozessoren ist bedeutend einfacher ist als die Programmierung derselben in einer Hardware-Beschreibungssprache. Die damit erreichbaren Geschwindigkeiten sind allerdings im Vergleich zu konventionellen Mikrocontrollern eher gering, da zuerst der Prozessor emuliert werden muss, der dann in einem zwei-

ten zeitlichen Vorgang das Anwendungsprogramm ausführt. Auch sind Soft-Core-Prozessoren aufgrund der um mindestens eine Grössenordnung geringeren Taktrate und der sehr einfachen Prozessorarchitektur derzeit noch keine Konkurrenz zu einem konventionellen Mikroprozessorchip, wie es z.B. in einem PC eingebaut ist. Gleichwohl gibt es viele Aufgabenstellungen, bei denen nicht höchste Rechenleistung das Ziel ist, sondern hohe Flexibilität und Anpassungsfähigkeit für kundenspezifische Lösungen. Beispiele dafür sind Steuergeräte im Automobil, Prozesssteuerungen und Regelungen industrieller Maschinen und Anlagen, sowie intelligente Haushalts- und Kommunikationsgeräte. Hier sind FPGAs gegenüber Mikroprozessoren bei eng umgrenzten Aufgabenstellungen auch kostenmäßig klar im Vorteil, da sie sowohl standardisierte Soft-Cores wie z.B. einen sog. Microblaze-Prozessor, als auch kunden- und anwendungsspezifische Logik auf einem einzigen Chip unterbringen können.

3.7 Grenzen der Softcore-Programmierung

Durch die substantielle Zunahme von Logikzellen auf dem Chip und damit auch der potentiell emulierbaren Soft-Cores wird das Problem der effizienten Nutzung von FPGAs allerdings auf die Software verlagert. Dieses Dilemma haben FPGAs mit Multi Core CPUs gemeinsam. Im Falle von Millionen von Logikzellen oder Hunderten von Soft-Cores pro FPGA-Chip ist es eine Herausforderung für den Programmierer, diese Ressourcen effektiv und effizient einzusetzen. Hinzu kommt: Nur wenn FPGAs direkt in VHDL oder Verilog programmiert werden, also keine oder nur wenige Soft-Core-Prozessoren verwenden, haben sie gegenüber Mikroprozessoren und Mikrocontrollern für die jeweils implementierte Funktion einen Geschwindigkeitsvorteil. Das war ein wichtiger Grund dafür, im CarRing II-Projekt zumindest die unteren ISO-Schichten nur in VHDL und nicht in C zu erstellen.

3.8 VHDL und Verilog

Obwohl in VHDL und Verilog Sprachelemente für parallele Programmierung gemäß der beiden Paradigmen von gemeinsamen Variablen und Botschaftenaustausch existieren, sind große parallele Anwendungen in diesen Sprachen bislang kaum realisierbar, da die Semantik paralleler VHDL Programme schwierig zu verstehen und noch schwerer zu testen ist. Der Grund dafür ist, dass VHDL und Verilog keine Hochsprachen sondern Hardware-Beschreibungssprachen sind. Hardware-Beschreibungssprachen bedeuten ein komplexes Prozedere, was den Programmentwurf und das Testen anbetrifft, sowie das Beachten vieler Regeln, anderfalls wird das VHDL- oder Verilog-Projekt scheitern, da kein synthetisierbarer Code erzeugt werden kann. Das bedeutet, dass ähnlich wie bei der Parallelrechner- oder bei der Cluster-Programmierung zu Beginn ein hohes Investment an Know How getätigt werden muss, bevor erste Erfolge eintreten. Im Unterschied zu Supercomputern und PC-Cluster sind FPGAs aber sehr klein, sehr preisgünstig und verbrauchen sehr wenig Strom, weshalb sich dieses Investment für bestimmte Anwendungen lohnt oder sogar der einzig gangbare Weg darstellt.

Der normale Vorgang bei der VHDL-Programmierung ist der, dass zuerst simuliert und dann synthetisiert wird, wobei eine erfolgreiche Simulation nicht automatisch zu einer erfolgreichen Chip-Synthese führt. Wenn bereits die Simulation nicht funktioniert, braucht die Synthese allerdings gar nicht erst versucht werden, weil ein Programmierfehler vorliegt. Umgekehrt gibt es

aber auch Fälle, bei denen eine Simulation nicht möglich ist, die Chip-Synthese aber erfolgreich ist. Insgesamt ist die Simulation ein nützliches Hilfsmittel für die Chip-Synthese, mehr aber auch nicht. Die Erstellung von synthesefähigem Code für ein FPGA ist deutlich komplizierter als die Simulation von VHDL, da der Simulator jedes VHDL/Verilog-Programm ausführen kann, dessen Syntax korrekt ist. Die Erfahrung zeigt, dass bei der Simulation vieles funktioniert, was sich später gar nicht oder nur auf Umwegen synthetisieren lässt. Ein Beispiel dafür ist die Verwendung einer `while`-Schleife zur Ausgabe von Werten auf einem Ausgangsport, die nur in der Simulation funktioniert. Dies liegt daran, dass VHDL ursprünglich ausschliesslich für Simulationszwecken entwickelt wurde, mit dem Ziel, damit Hardware zu beschreiben und diese vor derem Bau auf einem Rechner testen zu können. Erst später wurden Werkzeuge geschaffen, mit denen die Hardware-Beschreibungssprache VHDL zu einer Chip-Synthesesprache erweitert wurde. Des weiteren wurden integrierte Programmierumgebungen geschaffen, um den Chipdesignprozess zu vereinfachen und zu beschleunigen. Diese Synthesewerkzeuge sind aber als auch die Sprache „aufgesetzt“ zu betrachten und wurden beim ursprünglichen Entwurf des Sprachkonzepts von VHDL in keiner Weise berücksichtigt. Dadurch ist bei VHDL ein sehr komplexes semantisches Verhalten entstanden. Die Übernahme von `for` und `while` in den Sprachumfang der Synthesewerkzeuge, sowie die Kompatibilitätsprobleme zwischen den verschiedenen angebotenen Synthesewerkzeugen haben für weitere Schwierigkeiten bei der VHDL-Programmierung gesorgt.
Die Konsequenz aus dieser Entwicklung ist, dass VHDL/Verilog-Programmierung für einen Simulator einfach, für ein FPGA jedoch schwierig ist. Bemerkenswerterweise wird dies in den meisten Büchern und Tutorien über VHDL verschwiegen oder beschönigt. Dort wird stets der volle Sprachumfang beschrieben und oft so getan, als ob es keine Probleme gäbe. Viele VHDL-Programmierer beschränken sich deshalb nach ersten schlechten Erfahrungen auf die Simulation. Mit dem hier beschriebenen Software-orientierten Programmierstil kann dieses Dilemma teilweise behoben werden.

3.9 VHDL/Verilog versus System C

VHDL und Verilog existieren beide seit ungefähr 25 Jahren und sind in ihrer Mächtigkeit, aber auch in ihrer Komplexität ungefähr gleich hoch anzusetzen. Traditionell wird VHDL mehr in Europa und Verilog mehr in den USA verwendet. Als Alternative zu VHDL stehen im Prinzip noch einige C-Derivate wie System C [5] und Handel C [6] zur Verfügung. System C ist keine eigenständige Sprache, sondern eine Klassen-Bibliothek für C++, die diese Sprache um Objekte mit Methoden sowie um Makros erweitert. Die primäre Zielsetzung bei der Entwicklung von System C war, genau wie bei VHDL, die Modellierung und Simulation von Hardware. System C enthält dazu seinen eigenen Simulator in der Klassen-Bibliothek und ist deshalb ungefähr auf gleicher Ebene mit Modelsim anzusiedeln. Der Schritt zur Synthese wurde erst rel. spät und mit kleinen Schritten gegangen. Die Chipsynthese kann wie bei VHDL als „aufgesetzt“ betrachtet werden. Seit 1995 liegt eine IEEE-Richtlinie für System C vor (IEEE 1666-2005). Handel C ist eine eigene Sprache bestehend aus einer Teilmenge von C sowie Sprachkonstrukten, die die parallele Programmierung und das direkte Ansprechen von Hardware betreffen. Es gibt starke Ähnlichkeiten zur Sprache Occam, die in den 1980er Jahren verbreitet war. Bei beiden C-Derivaten werden Programme von einigen Programmierwerkzeugen zuerst in VHDL übersetzt und von dort aus wird mit den üblichen VHDL-Synthesewerk-

zeugen eine FPGA-Netzliste erzeugt. Aus den nachfolgenden Gründen haben wir aber davon Abstand genommen, System C und Handel C für das CarRing II-Projekt einzusetzen:

1.) Reale Codes in System C sind trotz des im Vergleich zu VHDL höheren Abstraktionsniveaus nicht sehr übersichtlich, was an der Vielzahl der zu benützenden Methoden und Macros liegt. Handel C schneidet hier besser ab.
2.) Bei beiden C-Derivaten handelt es um open source-Projekte aus dem universitären Umfeld mit den üblichen Problemen wie begrenzte Stabilität und Performanz und fehlender Support. System C und Handle C zur Chipsynthese werden nur in Einzelfällen von Firmen unterstützt, Handel C beispielsweise nur von Mentor Graphics [7]. Für VHDL existiert hingegen seitens der Industrie ein breites Angebot an Synthesewerkzeugen, Simulatoren und integrierten Entwicklungsumgebungen.
3.) VHDL ist ein ISO-Standard [4] und sehr weit verbreitet, was auf System C oder Handle C nicht zutrifft.
4.) Eine langdauernde Investion rechnet sich dann am meisten, wenn sich die Randbedingungen nur wenig ändern. Davon ist bei VHDL auszugehen bei C-Derivaten jedoch nicht.

3.10 Hardware-orientierter Stil versus Software-orientierter Stil

Obwohl VHDL und Verilog schon lange existieren, wurden in dieser Zeit bis auf Erweiterungen und Verbesserungen im Sprachumfang keine prinzipiellen Neuerungen im Programmierstil bekannt. Alle Lehrbücher [8][9][10][11][12][13][14][15][16] behandeln mehr oder weniger ähnliche Hardware-Beispiele. Der hier vorgestellte Programmierstil stellt deshalb auf dem Gebiet der FPGA-Programmierung eine eche Innovation dar. Der bekannte Hardware-orientierte Stil zeichnet sich dadurch aus, dass er u.a. die VHDL-Schlüsselworte `component`, `process`, `signal`, `port` und `port map` benützt. Damit kann man Schaltpläne der Digitaltechnik bestehend aus Flip Flops, Zählern, Vergleichern, Schieberegistern, Multiplizierern u.s.w. definieren und diese Hardware-Komponenten miteinander verbinden. In der Regel wird jedes der obigen Grundelemente der Digitaltechnik über einen VHDL-Prozess definiert und mit Hilfe eines oder mehrerer Signale mit anderen Grundelementen verbunden. Für genau dieses Anwendungsszenario ist der Hardware-orientierte Stil gut geeignet. Er wird nicht durch den Software-orientierte Stil verdrängt werden, da dessen Zielsetzung eine andere ist. Es wird deshalb den Hardware-orientierte Stil solange geben, wie es VHDL gibt.
Der Software-orientierte Stil dient dazu, mathematische und sonstige Algorithmen oder auch Rechnernetzprotokolle in VHDL zu implementieren. Das Ziel hierbei ist, VHDL als eine "normale" Programmiersprache wie z.B. C zu betrachten. Das ist sie jedoch nicht. Vielmehr hat VHDL gewisse Einschränkungen und Besonderheiten, die unbedingt beachtet werden müssen. Die Sprache bietet dafür aber auch mehr Freiheitsgrade als alle anderen prozeduralen oder objektorientierten Programmiersprachen. Der Software-orientierte Stil zeichnet sich u.a. dadurch aus, dass er die VHDL-Schlüsselworte `procedure` und `function` häufig verwendet. Grob ausgedrückt unterscheiden sich beide Stile in der Weise, wie sich eine VHDL `component` von einer `procedure` unterscheidet. Der Hardware-orientierte Stil ist in jedem Lehrbuch über VHDL dargestellt. Der Software-orientierte Stil ist neu und wurde vom Autor dieses Beitrags entwickelt. Im folgenden werden alle Aspekte und Regeln des Software-orientierten Stils vor-

gestellt. An einzelnen Stellen werden auch die beiden Stile miteinander verglichen, um deren Unterschiede zu verdeutlichen.

3.11 Zusammenfassung

Zusammenfassend kann gesagt werden, dass nicht nur auf dem Gebiet der klassischen Mikroprozessorprogrammierung, sondern auch auf dem Gebiet der FPGA-Programmierung eine immer größere Lücke zwischen dem klafft, was die Hardware leisten kann, und dem, was derzeit programmiertechnisch möglich ist. Deshalb wurde vom Autor ein algorithmischer Programmierstil für VHDL entwickelt, der als Software-orientierte VHDL-Programmierung bezeichnet wird. Der Stil kann 1:1 auf Verilog übertragen werden, da beide Sprachen ähnlich sind. Dieses neue Programmierparadigma benützt weniger Sprachelemente von VHDL als der klassische, Hardware-orientierte Stil folgt dabei aber anderen Prinzipien und arbeitet auf einem höheren Abstraktionsniveau, so dass unter dem Strich ein Gewinn entsteht. Die Anwendung ist einfach: auf einen Subset von VHDL müssen eine überschaubare Zahl von Programmierregeln angewendet werden. Das Neulernen von Spracherweiterungen ist nicht erforderlich.

4 Allgemeine Regeln des Software-orientierten Stils

4.1 Register oder Compiler-Interne Größe?

Die erste und entscheidende Frage beim Software-orientierten Stil ist, ob eine Variable vom Synthesewerkzeug in ein Register übersetzt wird, oder ob sie „nur“ eine Compiler-interne Zwischengröße darstellt. In ersterem Fall kann auf die Registervariable erst einen Takt später zugegriffen werden, d.h. das Programm ist an der Stelle, an der die Registervariable benützt wird, zunächst ein Mal logisch zu Ende und muss im zeitlichen Kontext, d.h. einen Takt später betrachtet werden. Im zweiten Fall (Variable ist eine reine Compiler-interne Zwischengröße) kann auf die Variable noch im selben Takt, d.h. bereits im nachfolgenden VHDL-Befehl lesend zugegriffen werden. Die Betrachtung eines Programms im zeitlichen Ablauf ist für den Hochsprachenprogrammierer ungewohnt und macht zu Beginn Schwierigkeiten. Damit verbunden ist auch die Frage nach der Ausführungsdauer einer Prozedur oder Funktion.
Wichtig ist nicht zu vergessen, dass Compiler-interne Zwischengrößen von allen Synthesewerkzeugen als Verdrahtung auf dem Chip implementiert werden. D.h., es gibt auf dem Chip einen Messpunkt, der die Variable repräsentiert. Solche Variablen sind ein Bündel von Verbindungen, bzw. im Falle einer binären Größe eine einzelne Verbindung. Funktionen, die Compiler-interne Zwischengrößen verarbeiten, werden durch kombinatorische Logik, d.h. durch Gatter, Multiplexer, Addierer, Multiplizierer etc., aber ohne speichernde Elemente synthetisiert. Dies gilt auch umgekehrt: kombinatorische Logik entsteht genau dann, wenn Compiler-interne Zwischengrößen verarbeitet werden sollen.

4.2 Regeln zur Registersynthese

Es gibt bestimmte Regeln für das Synthesewerkzeug, wann eine Variable als Register und wann sie als Compiler-Zwischengröße synthetisiert wird. Diese Regeln betreffen nur Pro-

grammvariablen, denn Signale werden stets als Register implementiert. Die Regeln für die Synthese von Registervariablen lauten:

1.) Wird auf eine Variable im Programm zuerst schreibend zugegriffen, bevor sie gelesen wird, braucht der Variablenwert nicht gespeichert werden, weil mit jedem neuen Aufruf des Prozesses oder der Prozedur, in der der Wert der Variablen definiert wird, der alte Wert überschrieben wird. Eine solche Variable wird nicht als Register synthetisiert, sondern als Compiler-Zwischengröße verbucht.
2.) Umgekehrt gilt: wenn die Variable zuerst gelesen wird, wird sie vom Compiler als Register implementiert, da ihr Wert sonst bis zum nächsten Prozess- oder Prozeduraufruf verloren ginge.
3.) Damit die Variable bereits beim ersten Lesen, d.h. beim ersten Programmaufruf einen sinnvollen Wert hat, muss sie eine Anfangswertinitialisierung erhalten. Dies erfolgt meist implizit mit Hilfe des „:=“-Operators. Bei einer Anfangswertinitialisierung über eine explizite Zuweisung liegt Fall 1) vor. Die Anfangswertinitialisierung mittels des „:=“-Operators ist aus der Sicht des Synthesewerkzeugs kein Schreiben, da der Wert der Variablen schon beim Laden des Programms auf das Chip entsteht und nicht erst bei der Programmusführung. In allen hier gezeigten Beispielprogrammen haben die Variablen eine Anfangswertinitialisierung, und es werden diejenigen Variablen, auf die mindestens eine der hier aufgeführten Syntheseregeln zutrifft, als Register synthetisiert. Ob eine Variable als Register tatsächlich synthetisiert wird, kann man anhand des Synthesеberichts des verwendeten Synthesewerkzeugs ermitteln.
4.) Wird in allen Zweigen eines `if ... then .... elsif ... else ...`-Sprachkonstrukts einer Variablen ein Wert zugewiesen, braucht die Variable vom Compiler ebenfalls nicht als Register implementiert zu werden, da sie bei jedem Aufruf des Prozesses- oder der Prozedur, in dem das Sprachkonstrukt enthalten ist, einen definierten Wert erhält.
5.) Umgekehrt gilt, dass bei `if` ohne `else` oder bei fehlender Spezifikation des Wertes der Variablen in mindestens einem der Zweige der Compiler die Variable als Register implementieren muss, damit sie auch beim nächsten Prozess- oder Prozeduraufruf noch wertmäßig vorhanden ist. Der Grund ist: sobald derjenige Zweig ausgeführt wird, in dem die Variable nicht spezifiziert ist, wird sie auch nicht geschrieben. Der nächste Aufruf würde deshalb bei einem lesenden Zugriff keinen aktuellen Variablenwert vorfinden.

Signale und Registervariable sollten im Software-orientierten Stil nur dann verwendet werden, wenn es unbedingt nötig ist, denn man kann nicht bereits im dem Takt auf ein Signal oder eine Registervariable lesend zugreifen, dessen Wert mittels einer Zuweisung soeben verändert wurde, sondern erst einen Takt später. Das macht die Verwendung von Signale und Registervariable schwieriger als bei Compiler-internen Zwischengrößen, da ein unmittelbares Lesen nach dem Schreiben zum Zwecke der Weiterverarbeitung der Registervariablen oder des Signals nicht nötig ist.
Darüberhinaus besteht bei einigen, aber nicht allen Synthesewerkzeugen, ein Unterschied bei den synthetisierten Grundelementen wie beispielsweise Zählern, Schieberegistern etc., wenn sie als Variable und nicht als Signal deklariert werden. Grundelemente mit Variablen werden bei diesen Synthesewerkzeugen umständlicher synthetisiert als mit Signalen. In nachfolgendem Beispiel:

```
signal i: integer range 0 to 7;
i <= i+1;
```

wird von dem Xilinx-Synthesewerkzeug xst ein echter Zähler synthetisiert. Während

```
variable i: integer range 0 to 7;
i := i+1;
```

als Zähler ohne eigentliche Zählfunktion, aber mit zusätzlichem Addierer, der +1 addiert, synthetisiert wird. Der Zähler ohne Funktion wirkt allerdings ähnlich wie ein Register, da er den aktuellen Zählerwert speichert. Darüberhinaus gibt es folgende wichtige Ausnahmen von obigen Registersyntheseregeln:

4.3 Ausnahmen bei der Registersynthese

Eine Variable wird dann nicht als Register synthetisiert,

1.) wenn nach einer Anfangswertinitialisierung stets nur lesend auf die Variable zugegriffen wird, d.h. wenn sie nie geschrieben wird. Aus der Variablen wird dadurch von der Funktion her eine Konstante, die dementsprechend auch als Konstante vom Synthesewerkzeug behandelt wird, was bedeutet, dass die Variable als ROM oder als Bündel von Drähten zu Vcc und GND implementiert wird.
2.) wenn durch die Verwendung der Variablen ein Grundelement definiert wird, das das Synthesewerkzeug kennt. Grundelemente bei xst beispielsweise sind Zähler, Akkumulatoren, Schieberegister, Addierer/Subtrahierer, Multiplizerer/Dividierer, Multiplexer, Dekodierer und Komparatoren. Bsp.: V1 := V1+1; wird nicht als Register sondern als Zähler plus Addierer synthetisiert, obwohl zuerst gelesen und dann geschrieben wird. Synplify von Synopsys hingegen erzeugt in diesem Fall stets ein Register mit Addierer.
3.) wenn das Synthesewerkzeug eine Optimierung vornimmt. Solche Optimierungen folgen Hersteller-internen Gesetzen, die nicht immer öffentlich bekannt sind. Deshalb kann man nur dann sicher sein, dass eine Variable tatsächlich als Register synthetisiert wird, wenn im Synthesebericht nichts Gegenteiliges steht. Im übrigen ist zu beachten, dass Zähler, Akkumulatoren und Schieberegister genau wie Register ebenfalls speichernde Elemente sind.

Beispiele für die Registersynthese sind in Code 1 in Form von Pseudocode bzw. echtem Code angegeben.

```
If Bedingung then
  var1 := 1; -- var 1 erhält den Wert 1
else
  var2 :=var1; -- var 1 wird nur gelesen
end if;
-- else-Zweig ohne Schreiben von var1 => var1 wird ein Register
-- if-Zweig ohne Schreiben von var2 => var2 wird ein Register
```

Programmcode 1: Beispiele für die Registersynthese in if then else.

4.4 Compiler-basierte Auswertung

Alle Synthesewerkzeuge versuchen aus Gründen der Ressourcenoptimierung auf dem FPGA-Chip bereits zur Compilezeit die Teile des Codes auszuwerten, die nicht zwingend die Synthese von Hardware auf dem Chip erfordern. Insbesonders kann es aus Optimierungsgründen vorkommen, dass Programme, in denen nur Variable als Compiler-interne Größen vorkommen und in denen kein Taktsignal verwendet wird, vollständig zur Compilezeit berechnet werden. Dies wird als Compiler-basierte Auswertung bezeichnet. Als Resultat werden die vom Compiler berechneten Ausgabegrößen als Anfangswertinitialisierung beim Programm-Download in das FPGA geladen. Das Chip gibt in diesem Falle die Ergebnisse nur noch aus, macht aber sonst keine Programmausführung mehr. In den Kapiteln zu for und while werden Beispiele für die Compiler-basierte Auswertung gezeigt. Die vollständige Berechnung eines Programms durch den Compiler ist allerdings eine Ausnahme, die nur unter bestimmten Bedingungen eintrifft. Ein Hinweis, dass diese Situation eingetreten ist, liefert die Compiler-Warnung „Warning, input clk is not used“, die besagt, dass trotz eines Taktsignals kein Register und auch kein anderes taktgesteuertes Grundelement synthetisiert wurde. Üblicherweise wird jedoch eine gemischte Auswertung und Ausführung des Programms in größerem Umfang durch das FPGA und in kleinerem Umfang durch den Compiler vorgenommen. Selbstverständlich gibt es aber auch VHDL-Programme, bei denen jedes einzelne Statement als Hardware synthetisiert wird. Grundsätzlich gilt, dass der Compiler nur die Programmteile synthetisiert, die unbedingt zu synthetisieren sind, alle sonstigen Teile versucht er selbst auszuwerten und zu berechnen. Das Problem, das bei dieser ressourcenschonenden Vorgehensweise auftritt, ist, dass VHDL-Programme scheinbar korrekt übersetzt werden, die in Wahrheit nicht synthetisierbar sind. Es wird i.d.R. höchstens eine Compiler-Warnung ausgegeben, der Compiler arbeitet aber weiter, und es wird eine FPGA-Netzliste erzeugt, so dass dem Programmierer sich der wahren Ursache der Meldung nicht immer voll bewusst wird. Insgesamt ist die Compiler-basierte Auswertung ähnlich „gutmütig“, wie es VHDL-Simulatoren sind. Bei Simulatoren für VHDL sind mehr und komplexere Sprachkonstruktionen möglich, als ein Synthesewerkzeug in eine Netzliste umsetzen kann. Wie gezeigt werden wird, ist aber unter bestimmten Bedingungen, die in den Kapiteln zu for und while aufgelistet werden, auch das Gegenteil möglich: der Compiler erzeugt eine korrekte Netzliste aus einem Programm, das ein Simulator nicht ausführen kann. Die Gründe für dieses Verhalten liegen beispielsweise darin begründet, dass Synthesewerkzeug und Simulator von verschiedenen Herstellern stammen können, so dass sich daraus abweichende Verhaltensweisen ergeben.

4.5 Notwendiger Ausgangsport

Einige Synthesewerkzeuge wie z.B. xst erfordern einen Ausgangsport, damit Sie überhaupt eine Netzliste für das FPGA erzeugen. Ein VHDL-Programm ist aus deren Sicht nur dann sinnvoll, wenn mindestens eine Ausgabe vom Programm vorgenommen wird. Deshalb wird in den hier dargestellten Codebeispielen das Ausgabesignal „NotwendigerAusgabeport“ in der Liste der Port-Signale aufgeführt und im Programm verwendet.

5 Spezielle Regeln des Software-orientierten Stils

5.1 Zähler

Beim Software-orientierten Programmierstil spielen Zähler eine entscheidende Rolle, da sie zur kontrollierten Ausführung der Programmbefehle durch Abzählen des FPGA-Taktes benötigt werden. In Code 2 wird deshalb ein Programmbeispiel für einen Zähler mit beliebigem Inkrement angegeben.

```
-- Package fuer Konstanten und Typdefinition:
package Konstanten is
-- Deklariere Konstanten
constant HoechsterWertVonZaehlvariable: integer := 15;
constant HoechsterWertVonInkrement: integer:= 7;
constant HoechsterWertAmAusgabeport: integer:= HoechsterWertVonZaehlvariable;
end Konstanten;


-- Hauptprogramm
-- Endlosschleife mit Registervariable
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;

entity UnbegrenzterZaehler is
Port (clk: in STD_lOGIC;
      NotwendigerAusgabeport: inout integer range 0 to
HoechsterWertAmAusgabeport);
end UnbegrenzterZaehler;

architecture One of UnbegrenzterZaehler is
  begin
    process(clk)
      variable Zaehlvariable: integer range 0 to
       HoechsterWertVonZaehlvariable := 9;
      variable Inkrement: integer range 0 to HoechsterWertVonInkrement := 3;
      begin
        if clk'event and clk = '1' then
          Zaehlvariable := Zaehlvariable + Inkrement;
          -- Der Befehl hat einen mehr oder weniger guten Zaehler zur Folge
          NotwendigerAusgabeport <= Zaehlvariable;
          -- Register. Wird einen Takt später gültig.
        end if; -- clk'event and clk = '1' ohne else
      end process;
  end One;
```

Programmcode 2: Beispiel für einen universellen Zähler.

Die Zählvariable startet in Code 2 mit dem Wert 9 und erhöht mit jedem FPGA-Takt den Wert um 3. Dabei läuft die Variable, die als 4-Bit-Größe deklariert ist, periodisch über. In Bild 1 wird das Syntheseergebnis, das xst erzeugt, dargestellt. Wie man sieht, wird ein Akkumulator erzeugt, der das Inkrement addiert, ein Addierer, sowie ein Register für das notwendige Ausgabesignal. Von den beiden erzeugten Komponenten Akkumulator und Addierer ist eine unnötig. Synplify hingegen generiert nur einen Addierer und zwei Register. Im Falle eines Inkrements

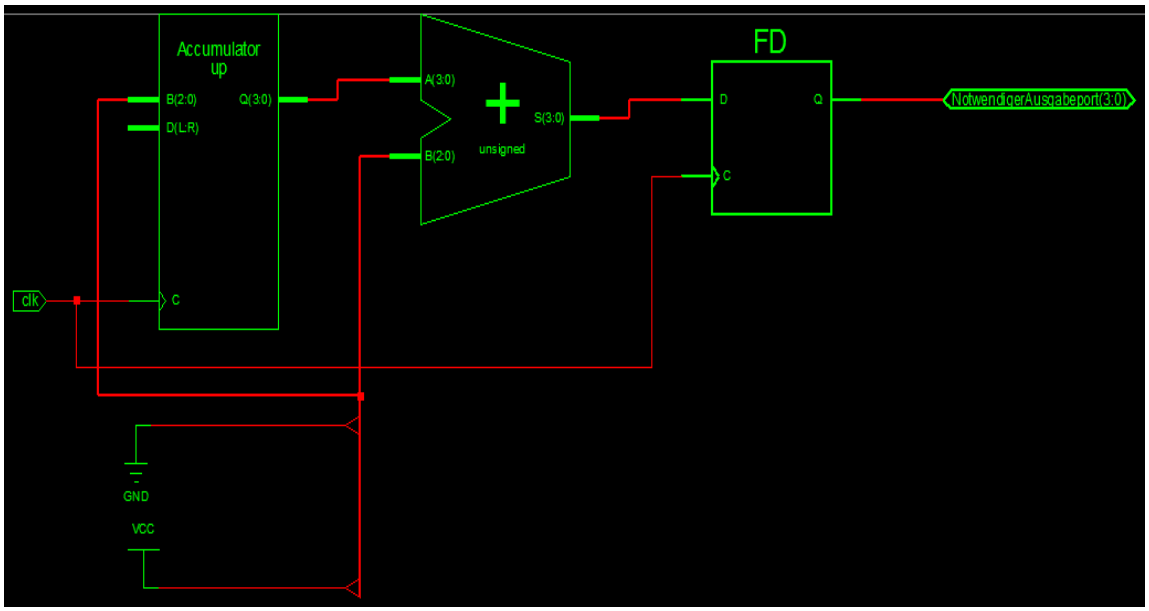

Bild 1: Syntheseergebnis von xst bei Code 2.

von Eins wird der Akkumulator vom xst-Synthesewerkzeug durch einen Aufwärtszähler ersetzt, bei Synplify gibt es keine Code-Änderung. Das bedeutet, dass das Ergebnis der Synthese von Code 2 davon abhängt, welches Synthesewerkzeug verwendet wird. Diese Erkenntnis lässt sich auf beliebige VHDL-Programme erweitern: sobald verschiedene Werkzeuge benützt werden, sind die erzeugten Netzlisten trotz gleichem Eingangsprogramm in der Regel verschieden,

Im Allgemeinfall lässt sich über Code 2 sagen, dass es bzgl. der erzeugten Netzlisten drei Kategorien von Synthesewerkzeuge gibt: in der ersten Kategorie wird erkannt, dass sich das Ausgangsregister für den notwendigen Ausgabeport und das speicherndes Element, das zählt, im wesentlichen gleich verhalten. Aus Optimierungsgründen wird deshalb das Ausgangsregister weggelassen und das Ausgangssignal direkt vom speichernden Element abgezweigt. Bei der zweiten Kategorie von Synthesewerkzeugen wird das speichernde Element, das zählt, und das Ausgaberegister in Serienschaltung gekoppelt, so dass eine Register-Pipeline entsteht. D.h., der Ausgang des Zählerregisters ist mit dem Eingang des Ausgaberegisters verbunden. Die Konsequenz daraus ist, dass bis zum Gültigwerden des Zählerausgangs die Dauer eines Taktes vergeht, und dass bis zum Gültigwerden der Ausgabe eine zweite Taktdauer benötigt wird, wodurch der Ausgabewert dem Zählerwert um einen Takt „hinterherhinkt“. Die dritte Kategorie von Synthesewerkzeugen schaltet den Eingang des speichernden Elements, das zählt, und den Eingang des Ausgaberegisters parallel. Deswegen, und weil beide Register denselben Takt verwenden, nehmen beide Registerausgänge zur selben Zeit denselben Wert an. Hier gibt es kein Hinterherhinken. Der dritte Fall darf jedoch nicht zu dem Fehlschluss verleiten, dass das Ausgaberegister keinen Takt an Zeit bis zum Gültigwerden seines Ausgangs benötigen würde, weil der Ausgang sofort dem Zählerwert folgt. Vielmehr gilt: Register benötigen stets ein Taktsignal. Je nachdem, wann der Eingang des Registers gültig wird, dauert es bis zu einem Takt, bis der Ausgang den Eingangwert annimmt. Diese Zeit muss abgewartet werden, sonst ist der Ausgang nicht aktuell.

Neben dem in Code 2 gezeigten Zähler gibt es noch weitere Möglichkeiten, Zähler zu erzeugen, die sich z.T. erheblich voneinander unterscheiden. Diese werden im Kapitel 8 "Programmschleifen beim Software-Orientierten Stil" genauer erläutert.

5.2 Prozedur und Funktion

Der Software-orientierte Programmierstil benützt u.a. die VHDL-Schlüsselworte `variable`, `procedure` und `function` aber nicht `component`. was wiederum vom Hardware-orientierten Stil verwendet wird. Beide Stile erlauben mit diesen Sprachelementen einen hierarchischen Programmentwurf, d.h. eine mehrfache Schachtelung von Code-Aufrufen. Man kann beispielsweise eine `component` deklarieren, die aus anderen VHDL-Komponenten besteht, und man kann eine `procedure` oder `function` schreiben, die andere Prozeduren oder Funktionen enthält, wobei bei der hier beschriebenen Methode zusätzliche programmiertechnische Maßnahmen erforderlich sind, die im Kapitel 10 "Geschachtelte Unterprogrammaufrufe im Software-Orientierten Stil" beschrieben werden. Grundsätzlich müssen in einem Software-orientierten VHDL-Programm stets zwei Konstrukte manuell hinzugefügt werden, die bei anderen prozeduralen Programmiersprachen nicht nötig sind, da VHDL ursprünglich nicht als Sprache zur Implementierung von Algorithmen entwickelt wurde. Diese Konstrukte sind ein Taktzähler und ein Abschnittszähler. Des weiteren gehören zum Software-orientierten Programmieren noch bestimmte Verhaltensregeln.

Im den nächsten Kapiteln werden der Takt-und Abschnittszähler erklärt und die Verhaltensregeln aufgelistet. Durch diese Maßnahmen, die nur einen geringen Verwaltungszusatz (=Overhead) in der Software darstellen, ist es möglich, VHDL ähnlich wie eine normale prozedurale Programmiersprache zu verwenden. Das bedeutet, dass jeder beliebige Algorithmus in Hardware auf einem FPGA-Chip implementiert werden kann, mit allen Vorteilen, die damit verbunden sind, wie z.B. die sehr hohe Geschwindigkeit bei der Programmausführung im Vergleich zu einer reinen Software-Lösung.

5.3 Taktzähler

Eine endliche Zahl von Schritten ist die Kernidee eines jeden Algorithmus, wie er in der theoretischen Informatik definiert ist. Beim Software-orientierten Programmierstil existiert deshalb ein global deklarierter Taktzähler, der jeden Schritt eines sequentiellen Algorithmusses abzählt, um Befehle kontrolliert, d.h. zu vorher festgelegten Zeitpunkten auszuführen. Die kontrollierte Befehlsausführung erfolgt durch Abzählen des FPGA-Takts in jeder Prozedur, Funktion oder Prozess und bewirkt, dass die Ausführungszeit derselben eine endliche Zahl vorher festgelegter Schritte umfasst. Als Konsequenz bedeutet dies, dass jeder Prozedur-, Funktions- und Prozessaufruf i.d.R. mehr als einen Takt Ausführungszeit benötigt. Eine Synthese rein kombinatorischer Logik findet nicht mehr statt.
Der notwendige Taktzähler kann auch implizit vorhanden sein, z.B. in Form eines Indizes für ein Feld (VHDL Array), der mit jedem Takt erhöht wird. Entscheidend ist, dass bei jedem (erneuten) Aufruf einer Funktion, einer Prozedur oder eines Prozesses mittels `if clk'event and clk='1'` oder mittels `wait until clk'event and clk='1'` klar ist, an welcher Stelle im Programm fortzufahren ist. D.h. es muss vom Programmierer stets festgelegt sein, wo das

FPGA bzgl. der Programmausführung fortfahren soll, sobald der Prozess, die Funktionen oder Prozeduren erneut aufgerufen wird. Diese Aufrufe erfolgen sehr oft, da die Bedingung `if clk'event and clk='1'` proportional zur Höhe des FPGA-Takts wahr wird.

5.4 Abschnittszähler

Ein Abschnittszähler, der dafür sorgt, dass große Algorithmen in kleinere Abschnitte zerlegt werden können, ist das zweite Konstrukt, das beim Software-orientierten Programmierstil wichtig ist. Solche Programmabschnitte werden im folgenden als Module bezeichnet. Der Grund für die Programmzerlegung ist, dass Synthesewerkzeuge nicht in der Lage sind, lange Algorithmen am Stück korrekt zu synthetisieren, oder zu viel Zeit dafür benötigen. Deshalb ist eine Zerlegung des Algorithmus in kurze Module mit bis ca. 1-10 Seiten VHDL-Code hilfreich. Der Abschnittszähler fügt die einzelnen Module wieder zusammen und sorgt für deren deterministische Ausführung, wobei Modul 1 vor Modul 2 ausgeführt wird, u.s.w. Der Abschnittszähler ist aus der Sicht des Synthesewerkzeuges ein endlicher Automat, der eine sequentielle Kette von Zuständen 1,2,..,n ohne Sprünge definiert. Im Zustand i des Automaten wird das Modul i ausgeführt. Innerhalb der Module kann der Steuerfluss, d.h. der Programmablauf beliebig sein, d.h., dort können und sollen bedingte und unbedingte Sprünge verwendet werden.
Zusammengefasst bedeutet dies, dass der Abschnittszähler die einzelnen Module in der gewünschten Reihenfolge aufruft. Der daraus resultierende endliche Automat stellt das Hauptprogramm dar. Oberhalb dieser Ebene gibt es pro Prozess keine weitere Steuerflussebene mehr.

6 Ergänzende Regeln des Software-orientierten Stils

6.1 Mnemonische Namensgebung

Aus Gründen der besseren Übersichtlichkeit werden die Namen der Zustände des Abschnittszählers und der Module so gewählt, dass sie für jeden Programmierer eine aussagekräftige und leicht zu merkende (=mnemonische) Bedeutung haben. Schlechter Programmierstil ist es, sie nur durchzunummerieren. Variable oder Signale werden in einem größeren Programm ebenfalls nicht mit V1, V2, ..., S1, S2, ... durchnummeriert, sondern haben mnemonische Namen. Erschwerend für eine mnemonische Namensgebung ist die Tatsache, dass VHDL Groß- und Kleinbuchstaben nicht unterscheidet.

6.2 Kein for und while

Die in jedem Modul enthaltenen Befehle müssen nach der Kompilation in dasjenige Repertoire von Grundelementen münden, das der Compiler sicher beherrscht, wie z.B. Zähler, Flip Flops, Multiplexer etc. Die VHDL-Befehle `for ... loop` und `while ... loop` gehören nicht zu den sicheren Befehlen, denn sie werden nur unter bestimmten Bedingungen richtig synthetisiert. Welche Bedingungen das sind, wird in Kapitel 11 "Besonderheiten des For-Befehls in

VHDL" und in Kapitel 12 "Besonderheiten des While-Befehls in VHDL" ausführlich erläutert, zusammen mit den Gründen, for und while nicht zu verwenden.

6.3 Viele Prozeduren und Funktionen

Jeder Abschnitt eines Algorithmus wird beim Software-orientierten Stil mit Hilfe einer `procedure` oder `function` implementiert, die wiederum Teil einer `procedure` oder `function` oder eines Prozesses sein können. Zwischen den Modulen können Daten ausgetauscht werden, indem diese als Aufruf- und Return-Parameter der Prozedur oder der Funktion deklariert werden. Wie bereits erwähnt gilt ferner, dass alle Prozeduren oder Funktionen rel. kurz sein müssen, um schnell und richtig synthetisiert werden zu können (max. ca. 10 Seiten Programm pro Prozedur oder Funktion).

6.4 Viele Module

Wichtig ist zu wissen, dass die Zahl der Module (=Abschnitte) nahezu beliebig groß sein darf. Tests hinsichtlich Chipverbrauch und Synthesezeit bei einem VHDL-Musterprogramm für den Fall von 4, 8, 16, 32 und 64 Abschnitten haben ergeben, dass der Chipverbrauch für den Software-Overhead des daraus notwendigen endlichen Automaten minimal ist, und dass die Synthesezeit bei kurzen Abschnitten im einstelligen Sekundenbereich bleibt, auch wenn viele Dutzend Module synthetisiert werden müssen. Viele und längere Abschnitte werden im einstelligen Minutenbereich synthetisiert. Zusammengefasst heißt dies, dass große und komplexe Algorithmen mit Hilfe mehrerer Abschnitte (Module) bestehend aus Prozeduren/Funktionen, die ihrererseits andere Prozeduren/Funktionen aufrufen, zu realisieren sind.

6.5 Mehrere Abschnittszähler

Der Software-orientierte Programmierstil erlaubt, genau wie der Hardware-orientierte Stil, Parallelverarbeitung und Interprozesskommunikation, indem Prozeduren oder Funktionen als Teile von parallel laufenden Prozessen miteinander Daten über Signale austauschen. Es können mehrere Abschnittszähler, d.h. mehrere endliche Automaten parallel verwendet werden, wenn mehr als ein Handlungsstrang nötig ist. Jeder Handlungsstrang wird als Prozess deklariert und ruft sequentiell seine Module auf. Sprünge in den parallel laufenden endlichen Autoamten sind nicht erlaubt. Innerhalb jedes Moduls können beliebige endliche Automaten auch mit Sprüngen deklariert werden.

6.6 Warten bis das Ergebnis fertiggestellt ist

Ergebniswerte von registerbehafteten Zuweisungen oder von registerbehafteten Grundelementen wie Zähler, Akkumulatoren und Schieberegistern liegen erst einen Takt später vor. Danach kann lesend auf sie zugegriffen werden, vorher wird der alte Wert gelesen. Dies muss unbedingt beachtet werden.

6.7 Konsequenzen bei Nichtverwendung eines Taktzählers und Abschnittszählers

Falls die oben beschriebenen Konstrukte Taktzähler und Abschnittszähler nicht verwendet, muss der Programmierer bestimmte Konsequenzen berücksichtigen, auf die im folgenden hingewiesen wird.

1.) Der Aufruf und die Ausführung jeder Funktion oder Prozedur kostet ohne Taktzähler stets weniger als einen Takt Zeit, da sie als kombinatorische Logik synthetisiert werden. Zuweisungen an Variable, die kein Register sind, werden nach der größten Summe an Gatterdurchlaufzeiten gültig, d.h. i.d.R. noch vor Ende eines Taktschlags.
2.) Falls in der Funktion oder Prozedur ein Register beschrieben wird, ist das Ergebnis erst einen Takt später gültig, D.h., das Ergebnis einer Funktion oder Prozedur liegt erst vor, nachdem diese bereits terminiert hat. Beim nächsten Aufruf der Funktion oder Prozedur kann der Ergebniswert dann verwendet werden.
3.) Falls in einer Funktion, einer Prozedur oder einem Prozess mehrere Register a, b, c, d, ... erzeugt und im weiteren Verlauf der Software so beschrieben werden, dass sie in einer logischen Kette gemäß b:=a; c:=b; d:=c; u.s.w. voneinander abhängen, werden alle Zuweisungen bereits mit dem nächsten Taktschlag gültig. Die Registerausgänge nehmen dabei die Werte an, der vor dem Gültigwerden der ersten Zuweisung an den Registereingängen anlagen. Das ist oftmals der Wert Null. Logisch abhängige Zuweisungen warten mit ihrer Ausführung also nicht darauf, bis an ihrem Eingang der richtige Wert anliegt. Es vergehen nicht mehrere Takte bis zum Ende der letzten Zuweisung. Dadurch werden alle Zuweisungen bis auf die erste sinnlos. Die logische Kette wird vom Synthesewerkzeug nicht in der Weise umgesetzt, wie es bei anderen Programmiersprachen üblich ist. Bei Funktionen und Prozeduren ist dieses Verhalten auch deshalb erforderlich, weil sie terminieren, bevor auch nur die erste Registerzuweisung ausgeführt ist. Bei taktsynchronen Prozessen ist das Verhalten notwendig, weil mit dem nächsten Takt der nächste Aufruf desselben Prozesses erfolgt. Aus der Sicht des Synthesewerkzeuges muss deshalb bis zum nächsten Takt alles abgearbeitet sein. Aufgrund dieser Problematik versuchen Synthesewerkzeuge zweierlei Strategien: Zum einen versuchen sie bereits zur Compile-Zeit die Kette auszuwerten, sofern dies möglich ist, und alle Register mit dem Wert von a zu initialisieren. Falls dies nicht möglich ist, versuchen sie zu vermeiden, dass b,c,d,... als Register synthetisiert werden, indem sie diese als Compiler-interne Größen verbuchen, die mit den Leitungs- und Gatterdurchlaufzeiten den Wert von a erhalten.

Es ist offensichtlich, dass es wegen der Konsequenzen 2 und 3 schwer ist, sinnvolle Programme in einem prozeduralen, d.h. üblichen Programmierstil zu schreiben. Bei Nichtverwendung eines Taktzählers ist nur der Hardware-orientierte Programmierstil möglich.

6.8 Deklarationen

Bei beiden Programmierstilen ist es wichtig, bei der Deklaration von Variablen, Signalen und Records den Wertebereich jeder Variablen und jedes Signals auf das Notwendige einzuschränken, da andernfalls unnötig viele Bits und Leitungen auf dem Chip erzeugt werden und zur

Verschwendung der Chip-Ressourcen führen. Die Begrenzung auf die tatsächlich notwendige Zahl von Bits pro deklariertem Objekt ist ein weiterer Unterschied zur reinen Hochsprachenprogrammierung, erfahrungsgemäß Schwierigkeiten macht, weil beispielsweise bei Zählern in while-Schleifen nicht berücksichtigt wird, dass ein numerischer Überlauf stattfinden kann.

6.9 Kommentare

Bei VHDL geschieht vieles implizit. Jeder VHDL-Befehl sollte deshalb mit mindestens einem sinnvollen Kommentar versehen werden, um sich selber und andere über die beabsichtigte Funktionen aufzuklären. Dies ist beim Software-orientierten Stil besonders wichtig, da die benötigten Operationen auf einem höheren Abstraktionsniveau beschrieben werden.

6.10 Gültigkeitsbereich von Variablen und Signalen

Wird eine Konstante, eine Variable oder ein Signal in verschiedenen Unterprogrammen jeweils für denselben Zweck verwendet, ist es besser, den Gültigkeitsbereich (scope) von Variablen, Signalen und Konstanten auszunutzen, diese in einem extra package zu deklarieren und dieses im übergeordneten Programm zu inkludieren. Dadurch entfallen gleichlautende Deklarationen in den einzelnen Unterprogrammen, und die Software wird übersichtlicher und kürzer.

6.11 Verkettete Zuweisungen und Operationen

Kausal verkettete Zuweisungen der Art b:=a; c:=b; d:=c; bedürfen aus zwei Gründen besonderer Aufmerksamkeit. Zum einen verhindern viele VHDL-Compiler in ihrer Optimierungsstufe Kettenzuweisungen, indem sie Zuweisungen, für die sie keine Register synthetisieren müssen, bereits zur Compilezeit auswerten und die verketteten Variablen beim FPGA Download mit ihren Endwerten initialisieren. Um eine Auswertung der logischen Kette nicht durch den Compiler, sondern durch das FPGA zu bewerkstelligen, kann entweder nach jeder Zuweisung der Variablenwert über einen port ausgegeben werden, oder alle Variablen werden durch Signale ersetzt. In beiden Fällen werden Register synthetisiert und dadurch eine compiler-basierte Auswertung verhindert.

Zum anderen wird eine logische Kette nur dann richtig synthetisiert und ausgeführt, wenn berücksichtigt wird, dass jede registerbehaftete Zuweisung oder Operation jeweils einen Takt Zeit bis zum gültig werden benötigt. Diese Wartezeit wird im nachfolgendem Beispiel (Code 3) durch ein case statement erreicht. Code 3 dauert vier Takte, und bei jedem Programmneustart wird der Taktzähler um Eins inkrementiert.

```
signal Taktzaehler : integer range 0 to 7 :=1; -- beginne mit Takt 1
signal a,b,c,d := integer range 0 to 7 :=0;
...
case Taktzaehler
  when 1 => b <= a;
    -- Register. Wird einen Takt später gültig.
  when 2 => c <= b; -- b ist hier gültig
    -- Register. Wird einen Takt später gültig.
  when 3 => d <= c; -- c ist hier gültig
```

```
    -- Register. Wird einen Takt später gültig.
  ...
end case;
```
Taktzaehler <= Taktzaehler+1; -- synth. Zaehler

Programmcode 3: Programmierung kausal verketteter Zuweisungen.

6.12 Initialisierung und Terminierung

Jeder Algorithmus der Informatik muss nach einer endlichen Zahl von Schritten terminieren. Dasselbe gilt für den Software-orientierten Programmierstil von VHDL. Die exakte Ermittlung der Zahl der benötigten Schritte ist somit ein wesentlicher Faktor bei der Software-orientierten Programmierung. Dies steht im Gegensatz zum Hardware-orientierten Stil, bei dem die Komponenten solange laufen, wie die Versorgungsspannung anliegt, d.h. alle Komponenten existieren hierbei zeitlich parallel und bis zum Ausschalten des FPGAs. Algorithmen hingegen starten zu einem deterministischen Zeitpunkt, arbeiten schrittweise und müssen nach einer gewissen Zeit terminieren, wenn sie korrekt sind. Das bedeutet, dass jedes VDHL-Programm eine Methode zur kontrollierten Programmbeendigung haben muss. Diese wird nach der erforderlichen Zahl von Schritten aufgerufen und stoppt das Programm. Bewährt hat sich hier, dass das letzte Modul, das aufgerufen wird, als einzigen Befehl das statement `null` enthält. Beispiel:

```
begin
  if clk'event and clk = '1' then
    if Abschnittszaehler = Abschnitt0 then
      ...
    elsif Abschnittszaehler = Ende then
      NULL; -- Mache nichts
    end if; -- Abschnittszaehler = Abschnitt0
  end if; -- clk'event and clk = '1'
```

Programmcode 4: Terminierung am Ende jedes Programms.

Vor jedem Programmstart steht die Initialisierung aller deklarierten Objekte (Variable, Signale, Records u.s.w.) mit Anfangswerten. Wenn diese nicht mit Hilfe des „:=“-Operators erfolgt, sondern mit einer expliziten Zuweisung, bedeutet dies für Variable, dass sie nicht als Register, sondern als Compiler-interne Größe synthetisiert werden, denn Registervariable werden nur dann synthetisiert, wenn die Variable zuerst gelesen und danach geschrieben wird. Damit das Lesen einer Variablen sinnvoll ist, muss sie zuerst initialisiert werden. Um diesen Widerspruch aufzulösen, ist bei Registervariablen nur die Anfangswertinitialisierung über den „:=“-Operators möglich. Ein Beispiel dafür ist in Code 5 gezeigt.

```
variable i: integer range 0 to HoechsterWertVoni := 1;
```

Programmcode 5: Anfangswertinitialisierung der Variablen i mit dem Wert 1.

Bei Signalen kann im Prinzip die Anfangswertinitialisierung auch über eine Signalzuweisung erfolgen. Allerdings gibt es hier die Einschränkung, dass eine Signalzuweisung zu Beginn der

Programmausführung verhindert, dass das Signal eine zweite Zuweisung im selben Programmdurchlauf erhalten darf, denn es wird von allen Zuweisungen an das Signal stets nur die letzte pro Programmdurchlauf ausgeführt. Alle vorangegangenen Zuweisungen werden ignoriert. Dies gilt für die Simulation und die Synthese.

Ohne Anfangswertinitialisierung wird von einigen Synthesewerkzeugen ein Initialisierung mit dem Wert Null vorgenommen. Andere Werkzeuge initialisieren mit der kleinstmöglichen Zahl im Wertebereich der Variablen. Um diese Inkompatibilitäten auszuschließen, ist es sehr ratsam, in jedem Programm eine definierte Anfangswertinitialisierung vorzunehmen. Im Falle einer compiler-internen Größe ist alternativ auch eine Wertzuweisung zur Laufzeit möglich. Wichtig ist zu wissen, dass der Compiler den Anfangswert, der über den := Operator zugewiesen wird, als Teil des FPGA-Programms generiert. D.h. der Anfangswert wird der Variablen zugewiesen, sobald das Programm in das FPGA geladen und ausgeführt wird. Für den Fall, dass das Programm bereits zur Übersetzungszeit vom Compiler ausgewertet wird (sog. Compiler-basierte Auswertung), ist diese Anfangswertinitialisierung deshalb wirkungslos, da das FPGA nichts mehr ausführt.

Im Standardfall, bei dem das FPGA das Programm ausführt, müssen bei Takt 0 alle deklarierten Objekte vom Programmierer einen definierten Anfangswert zugewiesen bekommen, dann folgen n Befehle oder n Sequenzen von taktlosen oder taktgesteuerten Befehlen. Diese werden jeweils an einem der k Programmtakte ausgeführt, wobei $n \leq k$ gilt. Danach terminiert die Software.
Auch beim Hardware-orientierten Stil, ist es ratsam, eine Anfangswertinitialisierung vorzunehmen. Diese kann mit dabei auch mit Hilfe eines Reset-Signals implementiert werden, das wiederum zu beliebigen Zeitpunkten aktiviert werden kann, aber im Falle eines taktsynchronen Resets nur zu diskreten Zeitpunkten wirksam wird.

6.13 Mehrfache Zuweisungen an Signale und Variable ohne if

Werden in einem seqentiellen Programmabschnitt (= Abschnitt ohne if) mehrfach hintereinander Werte an ein- und dasselbe Signal zugewiesen, bleibt nur die letzte Zuweisung übrig. Erfolgen mehrere Zuweisungen in einem seqentiellen Programmabschnitt an eine Variable, hängt es vom Synthesewerkzeug ab, was passiert. Deshalb sind mehrfache Zuweisungen an eine Variable oder ein Signal in einem seqentiellen Programmabschnitt unbedingt zu vermeiden. Muss sich z.B. für ein Timing-Diagramm der Wert eines Signals oder einer Variablen im Laufe der Zeit ändern, ist dafür ein EA mit Ein-/Ausgabe vorzusehen.

6.14 Mehrfache Zuweisungen an Signale und Variable mit if

Erhält eine Variable oder ein Signal in einigen oder allen Fallunterscheidungen einer case- oder einer if then else-Anweisung explizit verschiedene (numerische) Konstanten zugewiesen, ist das Syntheseergebnis oft überraschend. Explizite Konstantenzuweisung sind, um Überraschungen zu vermeiden, durch Zuweisungen von Elementen aus einem Enumeration Type an das Signal oder die Variable zu ersetzen. Warum das so ist, kann anhand des folgenden Beispiels erläutert werden:

Die Statusbits eines IO-Geräts soll von einem FPGA abgefragt werden, und je nach den Werten der Bits soll ein O.K. bzw. ein Fehlercode als numerischer Wert ausgegeben werden. Insgesamt seien drei verschiedene Ausgabewerte möglich: OK, Fehler1 und Fehler2, die durch die Zahlen 0,1 und 2 repräsentiert werden sollen. Deshalb wird das Signal/die Variable vom Programmierer als Integer im Bereich 0...2 deklariert, und OK erhält in den Programmzweigen den Wert 0, Fehler1 den Wert 1 und Fehler2 den Wert 2 explizit zugewiesen. Die dabei zugrunde liegende, implizite Annahme ist, dass 2 Bit für die Kodierung dieses Integers ausreichen.
Leider werden die Zuweisungen an das Ausgabesignal/die -variable entweder so synthetisiert, dass jeder Zahlenwert im FPGA fest verdrahtet ist und je nach Programmzweig über einen Multiplexer ausgewählt wird, oder es wird das Signal/ die Variable als Zustand eines EAs synthetisiert. Beides ist vom Programmierer nicht beabsichtigt worden und entsprechend sind die Konsequenzen: In beiden Fällen kann aus einem Integer, der als 2 Bit-Zahl geplant und deklariert war, ein Integer mit 3 Bit werden, denn im Falle des Multiplexers werden 3 Konstanten und ein 3->1 Multiplexers synthetisiert, und im Fall des EAs kann das Synthesewerkzeug eine one hot-Kodierung aus 3 Zuständen wählen. Wird anschliessend das Signal/die Variable mit Hilfe von Chipscope betrachtet, stimmen beoachtete Zahlenwerte und geschriebenes Programm nicht mehr überein!
Dieses Problem lässt sich nur so lösen, dass mehrfache Zuweisungen an Signale und Variable innerhalb von if oder case nicht explizit, sondern mit Hilfe der symbolischen Elemente eines Enumeration Types getätigt werden und dass im Synthesereport nachgeschlagen wird, welcher symbolische Wert welcher Zahl entspricht. Dieser Wert wird dann von Chipscope korrekt angezeigt.

7 Taktsynchrone Ausführung von Prozessen im Software-Orientierten Stil

Sobald in einem VHDL-Prozess eine Variable oder ein Signal als Register synthetisiert werden soll, wird ein Taktsignal für das Register benötigt. Um das Register erfolgreich zu synthetisieren, muss man einen taktsynchronen VHDL-Prozess definieren. Die Definition eines solchen Prozesses erfolgt durch Angabe eines Taktsignals (z.B. clock genannt) in der sensitivity list des Prozesses. Das Triggern des Registers mit diesem Takt erfolgt durch eine Abfrage zu Beginn des Prozess-Codes durch Angabe von:

```
if clock'event and clock='1' then
  A;
end if;
```

Alternativ ist auch ein `wait until clock'event and clock='1'` möglich. In beiden Fällen ist A eine Sequenz von mehreren taktgesteuerten, d.h. sequentiellen und taktlosen, d.h. rein kombinatorischen Anweisungen. Eine Anweisung ist dann taktgesteuert, wenn sie ein Register oder ein Schaltwerk zur Folge hat. Beispiele für Schaltwerke sind Zähler, Akkumulatoren und Schieberegister. Addierer, Multiplizier und Dividierer sind keine Schaltwerke sondern Schaltnetze. Durch „`if clock'event and clock='1' then A; end if;`“ wird der Dateneingang des Registers, das vom Synthesewerkzeug in A für ein Signal oder eine Variable synthetisiert

wurde, mit jeder ansteigenden Taktflanke abgetastet und der anliegende Wert im Register gespeichert. Genauso werden damit alle taktgesteuerten Grundelemente wie Zähler und Schieberegister angesteuert. Eine Anweisung ist dann ohne Verwendung eines Takts, wenn sie als Schaltnetz synthetisiert wird.

Leider wird durch eine der beiden obigen Abfragen aber auch der komplette Prozess mit jeder ansteigenden Taktflanke neu gestartet. Bei einer Taktrate von beispielsweise 50 MHz bedeutet dies 50 Mio. Neustarts desselben Programms in jeder Sekunde! Diese permanenten Prozessneustarts erschweren die Verwendung von VHDL als normale Programmiersprache erheblich und erfordern beim Software-orientierten Stil, dass die Programme „wiedereintrittsfähig“, d.h. reentrant geschrieben sein müssen. Reentrant heißt zum einen, dass nach jedem Programmneuaufruf klar sein muss, an welcher Stelle im Programm fortzufahren ist. Reentrant heißt zum anderen, dass sich mehrere gleichzeitig laufende Instanzen desselben Programms nicht gegenseitig stören dürfen. Eine Überlappung gleichzeitig laufende Instanzen desselben Programms ist dann ausgeschlossen, wenn eine Instanz terminiert, bevor die nächste Instanz des Programms startet.

7.1 Zählen und Abprüfen von Takten

Als Konsequenz aus der Wiedereintrittsfähigkeit der Software ergibt sich das bereits erwähnte Abzählen der Takte, um eine kontrollierte Aktivierung jedes Befehls zu ermöglichen. Das Zählen der Programmabschnitte erlaubt, neben der Feststellung, in welchem Abschnitt sich die Software gerade befindet, eine gröbere Untergliederung der Software in Module, die sicher synthetisiert werden können. Das Taktzählen und Abprüfen erlaubt, dass bei jedem Prozessneustart nur diejenigen VHDL-Befehle ausgeführt werden, die jeweils gewünscht sind und nicht beliebige andere. Das gezielte Aktivieren einzelner Befehle wird dergestalt bewerkstelligt, dass im Programm bei jedem taktgesteuerten Befehl auf den jeweiligen Zählerstand des Taktzählers Bezug genommen wird. Dies ist dann möglich, wenn ein Signal oder Variable als Synchronzähler existiert. Der Zähler speichert die Zahl der bereits ausgeführten Takte bis zum nächsten Aufruf, so dass der VHDL-Code nach jedem Programmneustart weiß, wieviele Befehle bereits ausgeführt wurden, und damit auch, welcher Befehl als nächstes auszuführen ist.

Das Beginnen an der richtigen Programmstelle nach jedem Programmaufruf aufgrund von `if clock'event and clock='1'` kann beispielsweise gemäß des in Code 6 angegebene Beispiels bewerkstelligt werden:

```
if clock'event and clock='1' then
  if Taktzaehler = 1 then A1; end if;
  if Taktzaehler = 2 then A2; end if;
  if Taktzaehler = 3 then A3; end if;
  ...
  Taktzaehler <= Taktzaehler+1; -- synth. Zaehler
end if;
```

oder durch:

```
if clock'event and clock='1' then
  case Taktzaehler is
    when 1 => A1;
    when 2 => A2;
    when 3 => A3;
  ...
 end case;
 Taktzaehler <= Taktzaehler+1; -- synth. Zaehler
end if;
```

Programmcode 6: .Beispiel für das richtige Fortfahren im Programm nach jedem Programmaufruf.

In allen Beispielen ist A1, A2, A3,... eine Sequenz aus taktgesteuerten und taktlosen (= rein kombinatorischen) Anweisungen. Jede Sequenz wird zu einem bestimmten Zeitpunkt gestartet, wobei die einzelnen Zeitpunkte sequentiell aufeinander folgen.
Darüberhinaus kann der Taktzähler auch als Index für eine Feldadressierung verwendet werden, wie dies in Code 7 gezeigt ist. In Code 7 wird erreicht, dass in jedem Takt ein Feldelement mit dem Wert des Taktzählers geschrieben wird. Dies erfolgt in aufsteigender Reihenfolge der Feldelemente.

```
Array1(Taktzaehler) := Taktzaehler;
Taktzaehler <= Taktzaehler+1;
```

Programmcode 7: Beispiel für die Verwendung des Taktzählers als Index und als Wert in einer Feldadressierung.

Ferner ist zu beachten, dass in Code 6 vorausgesetzt wird, dass die Ausführung von A_i nicht mehr als einen Takt Zeit benötigt (für alle i). D.h., es kann in obigen Beispielen nicht vorkommen, dass eine Ausführung von einem oder mehreren Befehlen in einem einzigen Takt versucht wird, obwohl dazu mehr als ein Takt an Ausführungszeit benötigt wird. Das heißt, dass in Code 6 die Summe der Ausführungszeiten aller Befehle beispielsweise im Programmabschnitt A1 die zur Verfügung stehende Taktdauer von einem Takt nicht überschreiten werden darf, sonst kommt es zu einer Überlappung von A1 mit A2 beim nächsten Prozessstart. Das muss zwar nicht unerwünschte Nebeneffekte haben, aber es kann. Das bedeutet für den Allgemeinfall, dass pro Prozessneustart nur so viel an taktgesteuerten und taktlosen Befehlen aktiviert werden darf, wie in diesem Takt auch abgearbeitet werden kann, anderfalls sind zwei oder mehr Instanzen von A_i gleichzeitig aktiv. Eine solche Situation muss ausgeschlossen werden. Sie kann sich dann ergeben, wenn im Prozess eine Prozedur oder eine Funktion aufgerufen wird und wenn dort die Summe der Gatterdurchlaufzeiten sehr groß ist, oder wenn in der Prozedur oder Funktion eine Taktzählung erfolgt und auf >1 Takt gezählt wird. Das Problem der Gatterdurchlaufzeiten kann durch Aufsummieren der in der Timing-Analyse des Synthesis Reports angegebenen Gatterschaltzeiten erkannt werden. Es muss hierbei derjenige Pfad durch den Graphen des Schaltplans gewählt werden, der die größte Länge hat. Dadurch wird die sog. worst case-Gatterdurchlaufzeit ermittelt. Die worst case-Gatterdurchlaufzeit kann oft durch Programmänderungen so verkleinert werden, dass es keine Überschneidung mit dem nächsten Funktions- Prozedur- oder Prozessaufruf gibt. In der Regel sind die Gatterdurchlaufzeiten in einem FPGA kleiner als die FPGA-Taktfrequenz bzw. Periodendauer, so dass ein stabiler (ein-

geschwungener) Zustand erreicht ist, bevor die Periodendauer endet. Will man allerdings die Geschwindigkeitsgrenzen des FPGAs erreichen, wird dieser Punkt von großer Bedeutung. Will man hingegen in einer Funktion oder Prozedur auf >1 Takt zählen, dann muss die im Kapitel 7.3 "Mehr als ein Takt Bearbeitungszeit pro Befehl oder Aufruf" beschriebene Methode verwendet werden. Dort wird gezeigt, wie die geschilderte Einschränkung aufgehoben wird, so dass A_i auf beliebig viele Takte Bearbeitungsdauer verallgemeinert werden kann.

7.2 Parallele Abarbeitung von Befehlen in einem Prozess

In vielen VHDL-Handbüchern und Sprachdefinitionen steht, dass alle Befehle in einem Prozess stets nacheinander ausgeführt werden würden. Wie unsere Tests ergeben haben, bedeutet dies nicht, dass von einem Befehl zum nächsten jeweils ein Takt Zeit vergeht. Es bedeutet leider auch nicht, dass eine sequentielle Abarbeitung der Befehle erfolgt. Die allseits gemachten Aussage sind unserer Ansicht nach für synthetisierten Code nicht zutreffend. Wenn Befehle im Programm durch kombinatorische Logik synthetisiert wird oder wenn Befehle im Programm durch Register bzw. registerbehaftete Grundelemente synthetisiert werden, dann wird ohne Verwendung des Software-orientierten Stile bereits in einem einzigen Takt eine größere Zahl solcher Befehle ausgeführt, da sie als parallel existierende Hardware-Komponenten auf dem Chip existieren.
Bei Registerzuweisungen gilt, dass man auf das Ergebnis der Registerzuweisung erst im nächsten Takt zugreifen kann. In diesem Sinne dauert jede Registerzuweisung stets einen Takt. Das bedeutet jedoch nicht, dass zwei aufeinanderfolgende Registerzuweisungen immer zwei Takte dauern. Im Gegenteil: Kommen n Registerzuweisungen bzw. registerbehaftete Grundelemente im Programmtext vor, werden sie vom FPGA gleichzeitig ausgeführt, obwohl sie nacheinander im Programmtext stehen. Der Grund dafür ist, dass alle Register und registerbehaftete Grundelemente denselben Chip-Takt benützen, d.h. sie werden im selben Takt gestartet und werden alle einen Takt später fertig. Beispielsweise benötigen die drei Befehle von Code 8 insgesamt nur einen Takt an Ausführungszeit.

```
i <= i+1;
j <=0;
k <= k+2;
```

Programmcode 8: Beispiel dreier Befehle, die voneinander unabhängig sind. Sie werden parallel ausgeführt.

Dies stellte eine Optimierung bzgl. der Ausführungszeit dar und ist dann kein Problem, wenn die Befehle nicht kausal abhängig sind. Ohne den Software-orientierten Stil würden aber auch kausal abhängige Befehle in einem einzigen Takt ausgeführt werden, wie bereits in Kapitel 6.7 "Konsequenzen bei Nichtverwendung eines Taktzählers und Abschnittszählers" gesagt wurde. Eine sequentielle Ausführung kausal abhängiger Befehle gibt es nur beim Software-orientierten Programmierstil. Die Befehle des Beispiels werden darüberhinaus vom Synthesewerkzeug mit anderen Befehlen, die nur als kombinatorische Logik synthetisiert sind, kombiniert, d.h. sie laufen parallel zu diesen. Bei kombinatorischen Befehlen hängt die genaue Zahl der in einem Taktschritt sicher ausführbaren Befehle von den Gatterdurchlaufzeiten und der FPGA-Taktfrequenz ab und ob sie in einer Kette voneinander abhängen. Man erfährt aus der Timing-Analyse

des Syntheseberichts, wie groß die summierten Gatterdurchlaufzeiten sind. Dies zu wissen, ist wichtig, da sie die Zahl der logisch abhängigen, kombinatorischen Befehle beschränken, die pro Prozess möglich sind.
Das gezielte Starten von Befehlen zu bestimmten Taktzeiten, das Überprüfen von deren Ausführungszeit, um eine Überlappung mit anderen Befehlen des nachfolgenden Prozessneustarts zu vermeiden, und die Berücksichtigung Chip-interner Parallelität erinnert an die Zeit der Mikroprogrammierung bei Rechnern. Im Falle von VHDL findet diese quasi auf Hochsprachenniveau statt, was erfahrungsgemäß bei der Programmerstellung Schwierigkeiten macht. Die zweite Schwierigkeit besteht darin, dass bei VHDL sehr vieles implizit geschieht. Dafür bietet VHDL aber auch mehr Möglichkeiten zur Programmgestaltung als alle anderen prozeduralen Programmiersprachen

7.3 Mehr als ein Takt Bearbeitungszeit pro Befehl oder Aufruf

Es gibt bei synthetisierbarem Code ein grundsätzliches Problem, wenn die Bearbeitungszeit eines Befehls oder der Aufruf eines Unterprogramms mehr als ein Takt beträgt. Der Grund dafür ist, dass sich in diesem Fall zwei oder mehr Instanzen desselben Codes zeitlich überlappen. Auf das FPGA-Chip bezogen bedeutet dies, dass auf dem Chip mehrere Chipteile gleichzeitig aktiv sind. Dies alleine wäre noch kein Problem, jedoch ist der Grund, warum ein Chipteil aktiv ist, jeweils ein anderer, nämlich ein bestimmter Neustart des Programms mit der potentiellen Gefahr unkontrollierbarer Überlappungen einzelner Programm- und/oder Chipteile. Dies ist auch der Grund, warum manche VHDL-Lehrbücher, wie z.B. [9] oder [14] dringend empfehlen, dass Unterprogramme nur kombinatorische Logik enthalten dürfen, weil so sichergestellt ist, dass deren Ausführungszeit weniger als ein Takt beträgt, da nur Gatterdurchlaufzeiten eine Rolle spielen. Es gibt jedoch nach unserer Auffassung zwei Methoden, mit denen mehr als ein Takt Bearbeitungszeit pro Befehl oder Aufruf. Die erste Methode besteht in der gezielten Abfrage eines Taktzählers, die zweite in der Verwendung von endlichen Automaten im rufenden und im gerufenen Programm, wobei das gerufene Programm den endlichen Automat des rufenden Programms am seiner seiner Ausführungszeit weiterschaltet. Beide Methoden sind für den Software-Orientierten Programmierstil unerlässlich.

7.3.1 Realisierung durch Abfrage eines Taktzählers

Diese Methode sollte dann angewandt werden, wenn es einen expliziten Taktzähler gibt, und wenn bei der Programmerstellung bereits klar ist, wie viele Takte für die Ausführung eines Befehl oder eines Unterprogramm benötigt werden.

Beispielsweise kann Code 6 kann auf einfache Weise so erweitert werden, so dass die Ausführungszeit jedes Befehls oder jeder Sequenz von Befehlen auf beliebig viele Takte erweitert werden kann. Dies geschieht in Code 9 durch:

```
if Taktzaehler = Taktwert1 then A1; end if;
if Taktzaehler = Taktwert2 then A2; end if;
if Taktzaehler = Taktwert3 then A2; end if;
...
Taktzaehler <= Taktzaehler+1; -- synth. Zaehler
```

oder durch:

```
case Taktzaehler
  when Taktwert1 => A1;
  when Taktwert2 => A2;
  when Taktwert3 => A3;
...
end case;
Taktzaehler <= Taktzaehler+1; -- synth. Zaehler
```

Programmcode 9: Beispiele für die Erweiterung von Programmteilen auf beliebig viele Takte Ausführungszeit.

In diesen Beispielen steht nicht nur ein einziger Takt für die Abarbeitung der Befehle in A_i zur Verfügung, sondern die Differenz Taktwert(i+1)-Taktwert(i), für alle i. Diese Differenz kann beliebig gewählt werden. Anstelle von Taktwertdifferenzen ist es auch möglich, ein Intervall von Takten anzugeben:

```
if Taktzaehler < Taktwert1 then ;
  A1
elsif (Taktzaehler >= Taktwert1) and (Taktzaehler < Taktwert2) then
  A2;
...
end if;
Taktzaehler <= Taktzaehler+1; -- synth. Zaehler
```

Programmcode 10: Beispiele für die Erweiterung auf beliebig viele Takten durch Angabe von Taktintervallen.

7.3.2 Realisierung durch gekoppelte endliche Automaten

Diese Methode muss dann angewandt werden, wenn es keinen expliziten Taktzähler gibt, oder wenn a priori nicht klar ist, wie viele Takte für die Ausführung eines Befehl oder eines Unterprogramms benötigt werden. Eine solche Situation tritt z.B. dann auf, wenn ein iterativer Algorithmus in VHDL implementiert wird, dessen Abbruchkriterium erst zur Laufzeit ausgewertet werden kann, oder wenn Status-Flags einer externen Hardware abgefragt werden, die sich irgendwann ändern können.

Die Methode beruht auf dem Ersetzen des Taktzählers durch gekoppelte endliche Automaten. Der Zählerzustand wird dadurch auf beliebige Systemzustände verallgemeinert. Das bedeutet für den Unterprogrammaufruf, dass sowohl das rufende als auch das gerrufene Programm über eigene endliche Automaten (EAs) verfügen müssen, die den momentanen Programmzustand bis zum nächsten Programmaufruf speichern. Wesentlich dabei ist, dass die Kopplung zwischen den EAs so ist, dass der EA des gerufenen Programms den EA des rufenden Programms weiterschaltet, sobald das gerufene seine Programmausführung beendet hat. Dadurch kann die Programmausführung beliebig lange dauern. Dieses Konzept kann auf einen ganzen Aufrufbaum aus n Unterprogrammen erweitert werden. Jedes gerufene Programm i im Aufrufbaum muss in einem eigenen EA(i) seinen momentanen Zustand speichern. Dadurch entsteht eine Menge aus Zuständen mit dem Ziel eines deterministischen Gesamtzustandes der Software. Die goldene Regel für diese Zustandsspeicher ist, dass mit jedem Taktschlag, d.h. mit jedem

Programmaufruf der Code an genau die Stelle zurückfinden muss, von der aus er mit dem Programm in korrekter Weise fortfahren kann.
Ein Beispiel: ein VHDL-Code bestehe aus einem Hauptprogramm und einem Unterprogramm, wobei das Hauptprogramm über einen endlichen Automaten namens EA1 und das Unterprogramm über einen endlichen Automaten namens EA2 verfügt. Die Zustände des EA1 sind:
`(DrucknopfAbwarten, LLIFInitialisieren, 1WortSenden, StatusPruefen, Stop).`

Die Zustände des EA2 lauten:
(PruefenLLIF, Einmalige_Datenuebergabe, SendePause1, HauptprogrammWeiterschalten)

EA1 und EA2 sind dafür zuständig, bei jedem Programmaufruf, d.h. mit jedem Taktschlag, externe Flags abzuprüfen und beim Eintreffen von boolschen Bedingungen den Zustand weiterzuschalten. Der letzte Zustand des EAs des Unterprogramms (`HauptprogrammWeiterschalten`) schaltet das Hauptprogramm in den nächsten Zustand. Bis zum Erreichen des Nachfolgezustands verharrt das Hauptprogramm an der Stelle, an der es das Unterprogramm aufgerufen hat. Die komplette Software ist in Code 26 dargestellt.

8 Programmschleifen beim Software-Orientierten Stil

Beim Software-orientierten Programmierstil werden Schleifen nicht mit Hilfe der Sprachkonstrukte for und while implementiert, sondern über einen Taktzähler, dessen Wert über ein if abgeprüft wird. Die Gründe liegen in mangelnder Kompatibilität der Synthesewerkzeuge bei for und while und in zu geringer bzw. abweichender Funktionalität dieser Befehle.
D.h., wenn man eine Schleife programmiert, die zur Laufzeit n Mal durchlaufen werden soll, ist dazu ein if-Befehl und ein Zähler erforderlich. Der Zähler zählt die Schleifendurchläufe, und der if-Befehl überwacht die Schleifenobergrenze. Ein Spezialfall sind Endlosschleifen, denn sie erfordern kein if. In Code 11 ist eine Endlosschleife mit Zählvariable dargestellt, und in darauffolgenden Beispiel (Code 12) wird eine Schleife mit einer endlichen Zahl von Durchläufen gezeigt. Dabei ist im Programmtext jeweils angegeben, wo der eigentliche Schleifenkörper, der mehrfach ausgeführt werden soll, eingebaut werden soll. Endlosschleifen werden hier deshalb gezeigt, weil sie besonders einfach sind und dadurch das Wesentliche leicht fassbar machen. Für die Implementierung (mathematischer) Algorithmen in Hardware haben sie keine Relevanz.

```
-- Endlosschleife mit Variable
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;

package Konstanten is
-- Deklariere Konstanten
constant HoechsterWertVoni: integer:= 3;
constant HoechsterWertAmAusgabeport: integer:= HoechsterWertVoni;
-- Begrenzung auf 2 Bit-Variable in diesem Beispiel
end Konstanten;

entity UnbegrenzterZaehler is
```

```
Port (clk: in STD_lOGIC;
      NotwendigerAusgabeport: inout integer
       range 0 to HoechsterWertAmAusgabeport);
end UnbegrenzterZaehler;

architecture One of UnbegrenzterZaehler is
  begin
    process(clk)
      variable i: integer range 0 to HoechsterWertVoni := 1;
      -- Zaehlvariable; startet mit 1 und zaehlt dann 1,2,3,0,1,2,3,0,...
      begin
        if clk'event and clk = '1' then
          i := i+1;
          -- Register => wird einen Takt spaeter gueltig
          -- Der Befehl hat einen mehr oder weniger guten Zaehler zur Folge

          -- Hier kann der eigentliche Schleifenkörper eingebaut werden

          NotwendigerAusgabeport <= i;
          -- Testausgabe. Es kann auch etwas anderes ausgegeben werden.
          -- Register. Wird einen Takt spaeter gueltig.
        end if; -- clk'event and clk = '1' ohne else
      end process;
  end One;
```

Programmcode 11: Endlosschleife in VHDL.

```
-- Endliche Schleife mit Variable
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;

package Konstanten is
-- Deklariere Konstanten
constant HoechsterWertVoni: integer:= 7;
constant HoechsterWertAmAusgabeport: integer:= HoechsterWertVoni;
-- Begrenzung auf 3 Bit-Variable in diesem Beispiel
type Abschnitte is (Abschnitt0, Stop); -- enumeration type
-- Programm hat nur einen Abschnitt + Stop
end Konstanten;

entity ifSchleife is
Port (clk: in STD_lOGIC;
      NotwendigerAusgabeport: inout integer
       range 0 to HoechsterWertAmAusgabeport);
end ifSchleife;

architecture One of ifSchleife is
  signal Abschnittszaehler: Abschnitte:= Abschnitt0;
  -- starte mit Abschnitt0
  begin
    process(clk)
      variable i: integer range 0 to HoechsterWertVoni := 1;
      -- startet mit 1 und zaehlt dann 1,2,3,4,5,6,7 und stoppt. D.h., es
      -- werden 7 Schleifendurchgänge gemacht.
      begin
        if clk'event and clk = '1' then
          if Abschnittszaehler = Abschnitt0 then
            -- Beginn der if-Schleife
```

```
        if i<HoechsterWertVoni then
          i := i+1;
          -- Register => wird einen Takt spaeter gueltig
          -- Der Befehl hat einen mehr oder weniger guten Zaehler
          -- zur Folge

          -- Hier kann der eigentliche Schleifenkörper eingebaut werden

        else -- i>=HoechsterWertVoni
          Abschnittszaehler <= Stop;
           -- Register. Wird einen Takt spaeter gueltig.
        end if; -- if-Schleife
        NotwendigerAusgabeport <= i;
        -- Register => wird einen Takt spaeter gueltig
        -- Ausgabe ist ausserhalb der Schleife am Schluss.
        -- Testausgabe. Es kann auch etwas anderes ausgegeben werden.
      else -- Abschnittszaehler <= Stop
        -- Programmende erreicht
        null;
      end if; -- Abschnittszaehler = Abschnitt0
    end if; -- clk'event and clk = '1' ohne else
  end process;
end One;
```

Programmcode 12: if-Schleife mit endlich vielen Durchläufen.

Es gibt erhebliche Unterschiede, wie die gezeigten Schleifen von den verschiedenen Syntheseswerkzeugen übersetzt werden. Des weiteren gibt es gravierende Unterschiede im Kompilat wenn anstelle einer Zählvariablen ein Zählsignal verwendet wird. Im folgenden wird hinsichtlich dieser beiden Aspekte xst von Xilinx mit Synplify von Synopsis verglichen, wobei beide Varianten (Variable und Signal als Zähler) verwendet werden.

8.1 Synplify-Schleife mit Endloszähler und Zählvariable

In Bild 2 ist das von Synplify erzeugte Syntheseergebnis für den in Code 11 dargestellten Endloszähler mit Zählvariable gezeigt. Die Zählfunktion wird über einen rückgekoppelten Addie-

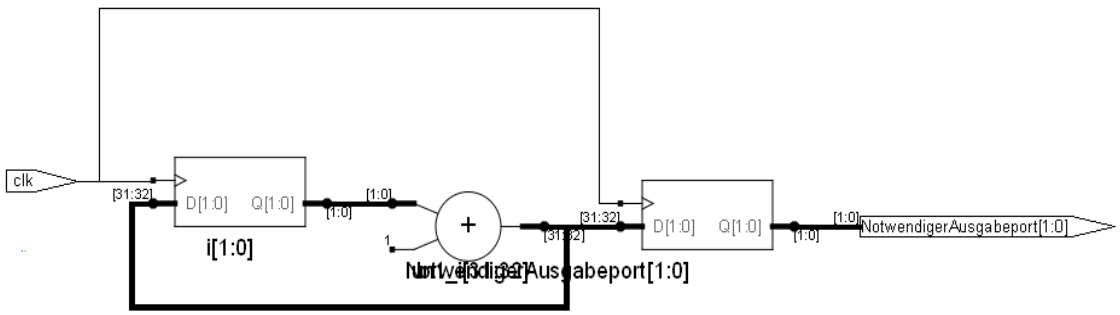

Bild 2: Von Synplify erzeugtes Kompilat für den in Code 11 dargestellten Endloszähler.

rer am Ausgang des Variablenregisters implementiert. Das Ausgangssignal erhält ein zweites Register, das denselben Takt und denselben Input wie die Zählvariable verwendet, d.h. es wird noch im selben Takt gültig und ändert sich synchron mit der Zählvariablen. Ein extra Register für das Signal wäre deshalb nicht notwendig. Der von Synopsys erzeugte Code ist aus diesem Grunde nicht optimal. Das Verhalten des Endloszählers ist in Bild 3 gezeigt. Bis auf die Anfangswerte verhalten sich i und das notwendige Ausgabesignal wie erwartet gleich. Es ist

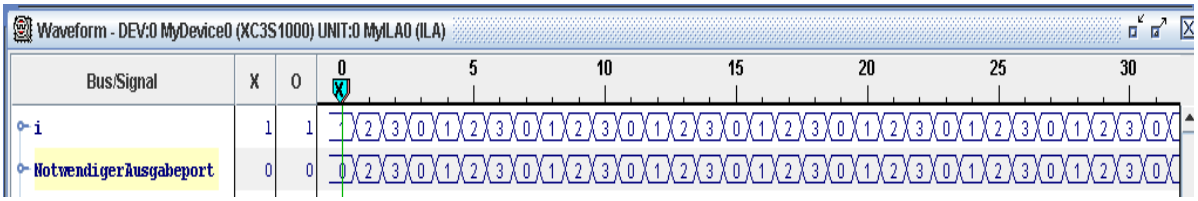

Bild 3: Verhalten des Endloszählers von Code 11 auf dem FPGA.

noch bemerkenswert, dass dieselbe Hardware synthetisiert wird, wenn statt i := i+1; der Befehl i := i-1; oder i := i+inkrement; eingesetzt wird, nur die Konstante, die vom Addierer hinzugezählt wird, ändert sich.

8.2 Synplify-Schleife mit Endloszähler und Zählsignal

Wird anstelle einer Zählvariable ein Zählsignal im Programmtext verwendet, wobei alles andere gleich bleibt, ändert sich das Kompilationsergebnis in signifikanter Weise. In Bild 4 ist das von Synplify erzeugte Kompilat dargestellt. Die Zählfunktion wird jetzt über einen rückge-

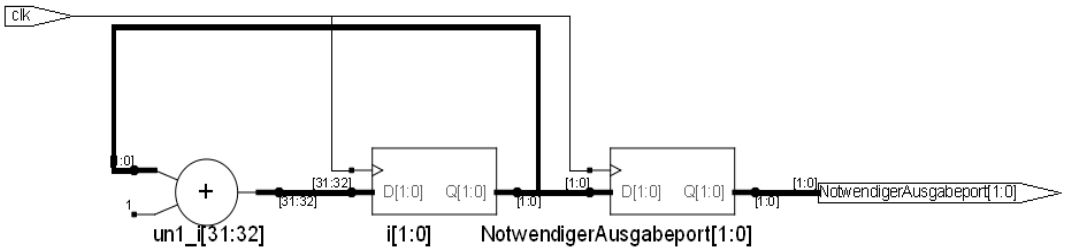

Bild 4: Von Synplify erzeugtes Kompilat für den in Code 11 dargestellten Endloszählers mit Zählsignal anstelle einer Zählvariablen.

koppelten Addierer am Eingang des Registers implementiert, d.h. die Position des Addierers hat sich vom Ausgang des Registers zum Eingang verschoben. Das Ausgangssignal erhält, wie immer, ein eigenes Register. Dieses bildet mit dem Zählregister eine Pipeline, d.h. sein Ausgangswert wird erst einen Takt später als der Zählerwert gültig. Das neue zeitliche Verhalten ist in Bild 5 gezeigt.

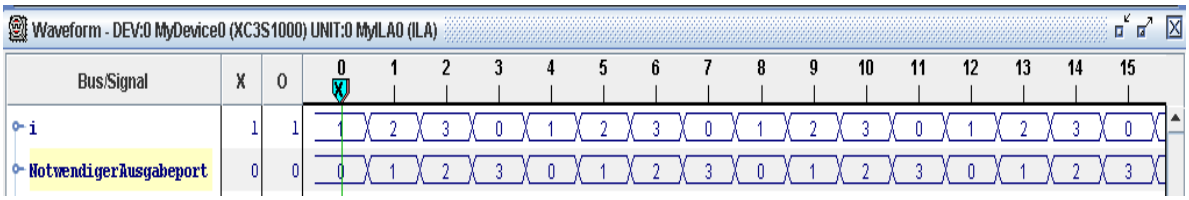

Bild 5: Verhalten des in Code 11 dargestellten Endloszählers auf dem FPGA mit Zählsignal anstelle einer Zählvariablen.

8.3 xst-Schleife mit Endloszähler und Zählvariable

Im Vergleich dazu folgen die Syntheseergebnisse von xst für dieselben Programmbeispiele. In Bild 6 ist das von xst erzeugtes Kompilat gezeigt. Die Zählfunktion wird über einen Aufwärts-

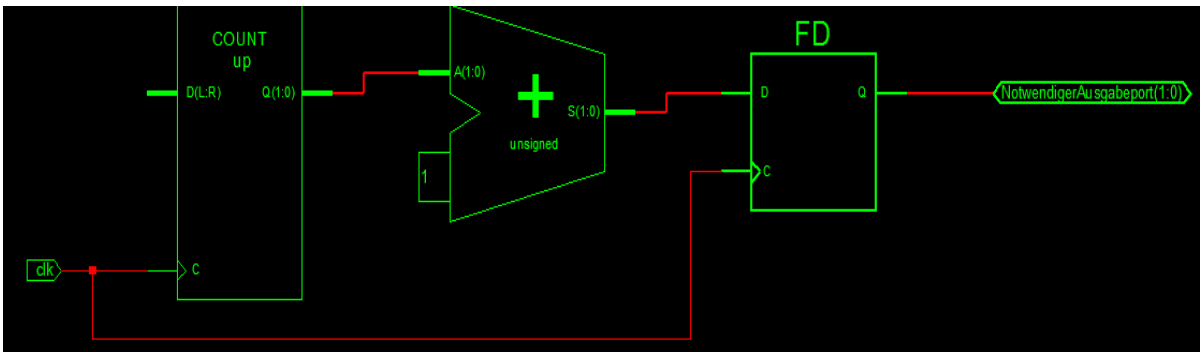

Bild 6: Von xst erzeugtes Kompilat für den in Code 11 dargestellten Endloszähler.

zähler und einen Addierer, der zum Zählerwert +1 hinzuzählt, implementiert. Der Addierer hat bis auf die Bereitstellung des Anfangswertes von 1 keine sinnvolle Funktion. Das Ausgangssignal verfügt wie bereits zuvor über ein eigenes Register. Dieses bildet die bereits in Bild 4 gezeigte Pipeline mit dem Zählsignal. Das zeitliche Verhalten ist analog wie in Bild 5.
Insgesamt ist der von xst erzeugte Code in diesem Falle schlechter als der von Synplify, da der Aufwand für einen Zähler mit Addierer über dem eines Registers mit Addierer liegt. Außerdem arbeitet die erzeugte Schaltung zeitverzögert im Vergleich zu Synplify, da beim xst-Ausgabesignal zwei Takte bis zur ersten Aktualisierung des Werts vergehen. Andere Erfahrung mit xst zeigen, dass dieses Synthesewerkzeug für den Fall, dass Variable im Programmtext verwendet werden, nicht in der Lage ist, guten Code zu synthetisieren. Bei ausschließlicher Verwendung von Signalen gibt es aber Fälle, in denen xst besseren Code als Synplify erzeugt.

8.4 xst-Schleife Endloszähler und Zählsignal

In Bild 7 ist das von xst erzeugtes Kompilat für den Endloszähler mit Zählsignal anstelle einer Zählvariablen gezeigt. Die Zählfunktion wird jetzt nur über einen Aufwärtszähler implementiert. Der Aufwand ist dadurch geringer, und der erzeugte Code trifft genau den beabsichtigten Zweck, so dass dieses Ergebnis besser als bei Synplify ist. Allerdings ist das Register für das

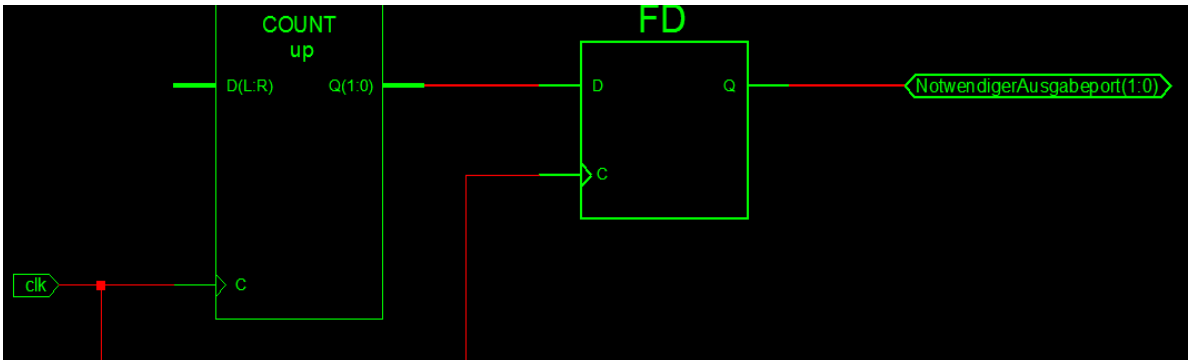

Bild 7: Von xst erzeugtes Kompilat für den in Code 11 dargestellten Endloszähler.

Ausgangssignal ohne Funktion, da bereits der Zähler den Ausgabewert bis zum jeweils nächsten Takt speichert. D.h., auch das xst-Kompilat ist nicht optimal.

8.5 Synplify-Schleife mit endlichem if-Zähler und Zählvariable

In Bild 8 ist das von Synplify erzeugtes Kompilat für den in Code 12 dargestellten endlichen Zähler mit Zählvariable gezeigt. Die Zählfunktion wird genau wie im Falle der Endlosschleife

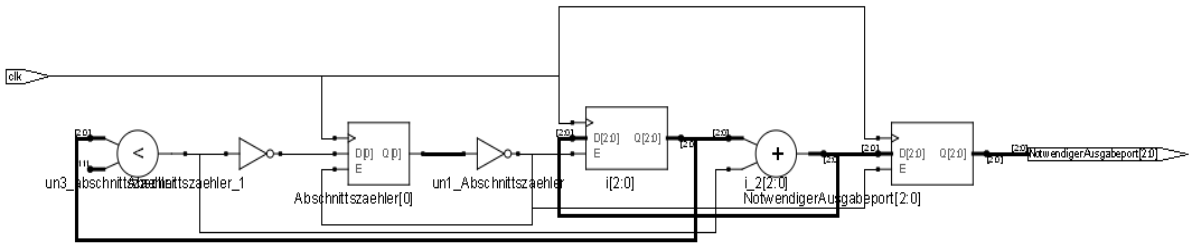

Bild 8: Von Synplify erzeugtes Kompilat für den in Code 12 dargestellten endlichen Zähler.

mit Variable über einen rückgekoppelten Addierer am Ausgang des Zählregisters implementiert. Synplify macht an dieser Stelle keinen Unterschied zwischen endlosem und endlichem Zähler. Das Ausgangssignal erhält ein eigenes Register das redundant ist. Die endliche Zählfunktion wird über einen endlichen Automaten realisiert, der den Zähler nach Erreichen der Schleifenobergrenze über dessen Enable-Eingang stilllegt. Der Automat benützt einen Vergleicher (Komparator), der den Zählerwert mit der Schleifenobergrenze vergleicht. Insgesamt ist der erzeugte Schaltplan nicht optimal im Sinne minimalen Aufwandes, aber er ist verständlich, da aus einfacheren Grundelementen aufgebaut. Das daraus resultierende zeitliche Verhalten ist in Bild 9 gezeigt. Bis auf die Anfangswerte sind i und das notwendige Ausgabesignal identisch.

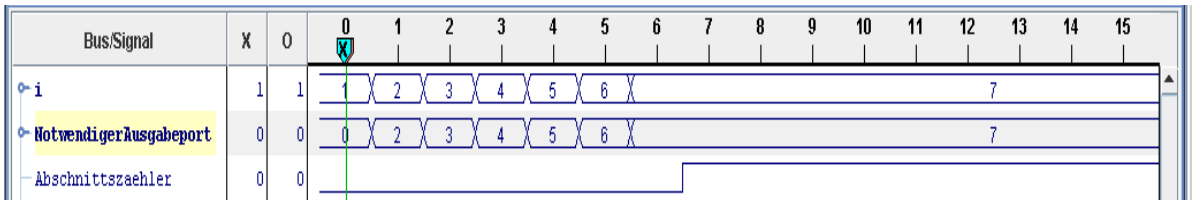

Bild 9: Verhalten des endlichen Zählers auf dem FPGA.

8.6 Synplify-Schleife mit endlichem if-Zähler und Zählsignal

Wird anstelle einer Zählvariable ein Zählsignal erzeugt, wobei alles andere gleich bleibt, ändert sich das Kompilationsergebnis genau wie beim Endloszähler (Bild 10). Die Zählfunk-

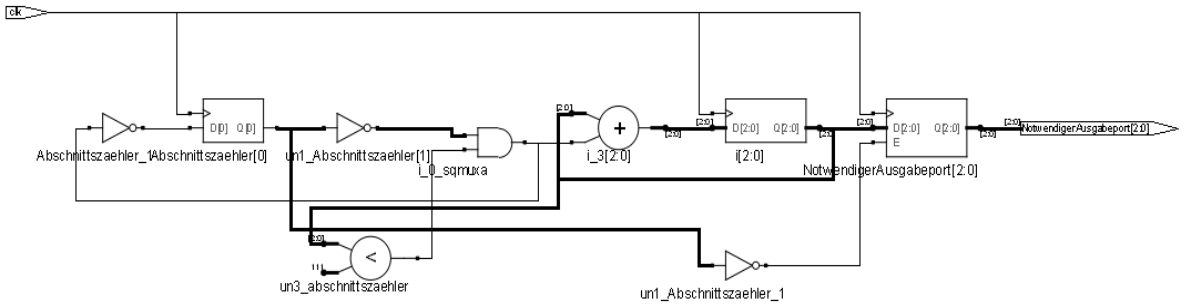

Bild 10: Von Synplify erzeugtes Kompilat für den in Code 12 dargestellten endlichen Zähler mit Zählsignal anstelle einer Zählvariablen.

tion wird, wie im Beispiel von Code 11, über einen rückgekoppelten Addierer am Ausgang des Registers implementiert. Das Ausgangssignal erhält ein eigenes Register und bildet eine Pipeline. Insgesamt benötigt Synplify ein Und-Gatter und einen Inverter mehr als im Falle der Zählvariablen, und der erzeugte Schaltplan ist bereits rel. umfangreich.

8.7 xst-Schleife mit endlichem if-Zähler und Zählvariable

Bild 11 zeigt das von xst erzeugtes Kompilat für einen endlichen Zähler mit Zählvariable. Der erzeugte Schaltplan ist deutlich unübersichtlicher als bei Synplify. Die Zählfunktion wird über einen Addierer realisiert, der zum Ausgabesignal solange +1 addiert, wie ein endlicher Automat (realisiert im unteren Flip Flop) angibt. Ein Register zum Speichern des Zählerwertes gibt es nicht. Diese Funktion wird vom Ausgangsregister (Flip Flop oben rechts) mit übernommen, was eine Ressourcenoptimierung darstellt.

Die Schleifenobergrenze wird über einen „Less Than“-Komparator abgeprüft. Ist sie erreicht, wird das Register des Ausgangssignals über seinen Clock Enable-Eingang desaktiviert. Unnötigerweise erzeugt xst zusätzlich einen aufwändigen Multiplexer, um dem Ausgangsregister den richtigen Eingabewert in Abhängigkeit der Zustände „Zählen“ oder „Stoppen“ zuzuführen. Hier ist Synplify deutlich besser. Das resultierende Verhalten ist dennoch korrekt und in Bild 12 dargestellt.

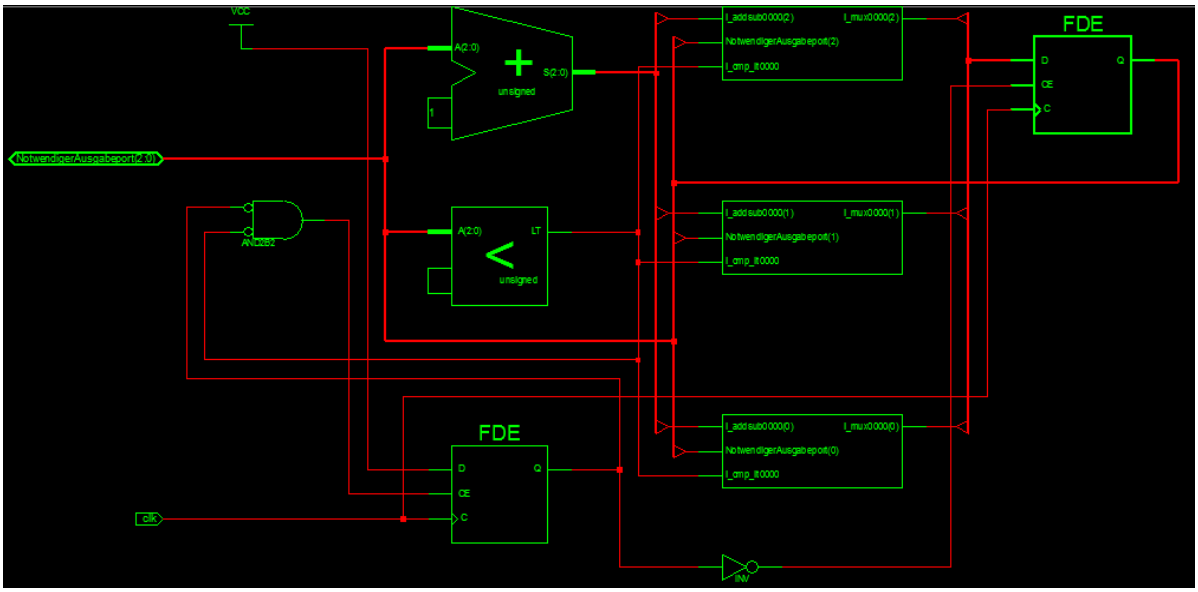

Bild 11: Von xst erzeugtes Kompilat für den in Code 3 dargestellten endlichen Zählers.

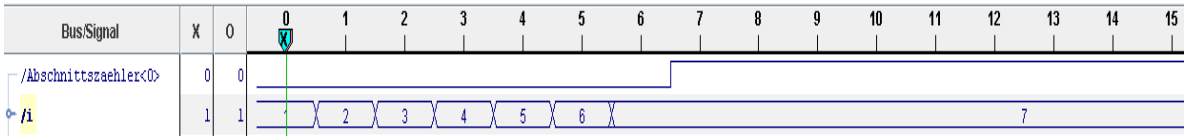

Bild 12: Verhalten des endlichen Zählers auf dem FPGA.

8.8 xst-Schleife mit endlichem if-Zähler und Zählsignal

Wird anstelle einer Zählvariable ein Zählsignal erzeugt, ändert sich das Kompilationsergebnis in der in Bild 13 dargestellten Weise und wird wesentlich übersichtlicher. Die Zählfunktion

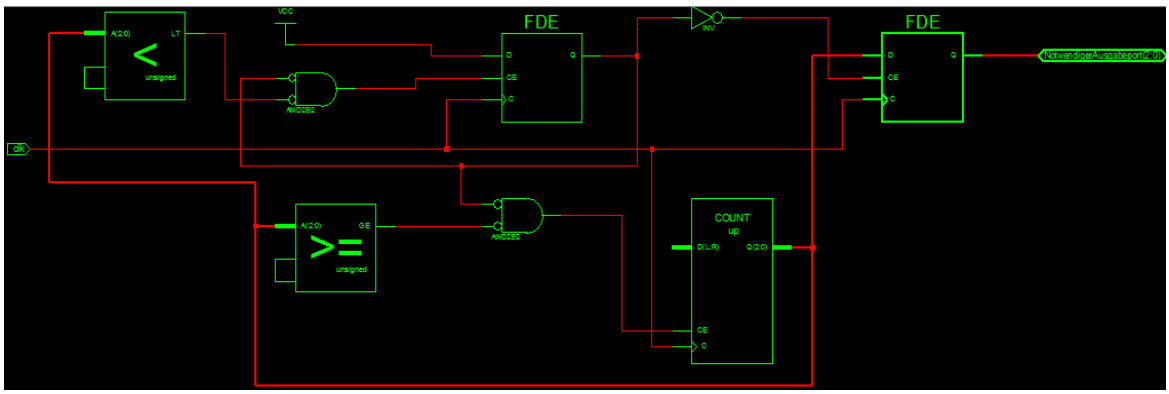

Bild 13: Von xst erzeugtes Kompilat für den in Code 12 dargestellten endlichem Zähler mit Zählsignal anstelle einer Zählvariablen.

wird über einen Zähler realisiert, und das Ausgangssignal hat ein Register. Allerdings benötigt xst jetzt zwei Komparatoren. Der obere Komparator dient dem endlichen Automaten (realisiert

im oberen linken FF) zur Abprüfung der Schleifenoberobergrenze. Der untere Komparator wird vom Zähler für denselben Zweck benötigt, ist also redundant. Trotzdem ist der erzeugte Schaltplan deutlich besser als im Falle der Zählvariablen. Das zeitliche Verhalten ist in Bild 14 gezeigt.

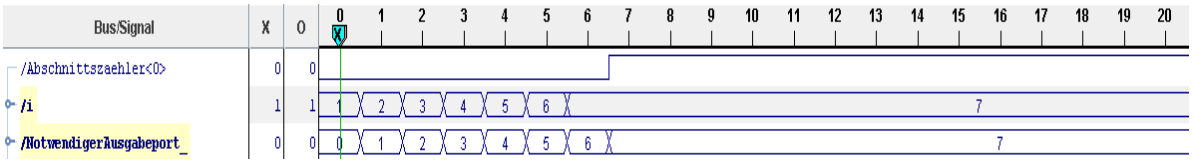

Bild 14: Verhalten des in Code 12 dargestellten endlichen Zählers auf dem FPGA mit Zählsignal anstelle einer Zählvariablen.

8.9 Zusammenfassung

Zusammenfassend ist zu sagen, dass Programmschleifen im Software-orientierten Programmierstil mit Hilfe von if und einem Schleifendurchgangszähler implementiert werden. For und while werden aus Gründen mangelnder Kompatibilität und zu geringer Funktionalität nicht verwendet. Weiterhin unterscheiden sich if-Programmschleifen Synplify und xst im erzeugten Kompilat stark voneinander, wobei Synplify besser als xst abschneidet, was die Übersichtlichkeit des erzeugten Schaltplans und die Schonung der Ressourcen anbetrifft. Diese Aussage lässt sich auch auf andere Sprachkonstrukte verallgemeinern. Die einfache Version von XST hat den Vorteil, kostenfrei zu sein, es ist aber nicht das beste Synthesewerkzeug. Allerdings ist die graphische Darstellung des generierten Schaltplans bei xst viel besser als bei Synplify, jedoch kommt es vor, dass Leitungen im Schaltplan fehlen, was schwerer wiegt als eine ansprechende Optik. Das Fehlen von Leitungen im Schaltplan trat darüberhinaus auch bei der von uns verwendeten kostenpflichtigen xst-Version auf.

Im weiteren wird anhand von drei VHDL-Beispielen gezeigt, wie Software-orientierte Programmierung im Prinzip abläuft. Diese Beispiele zeigen, wie die zuvor erläuterte Theorie in der Praxis umgesetzt wird.

9 Programmierbeispiele

9.1 Abschnitts- und Taktzählung und deren Abprüfung

Im nachfolgenden Beispiel (Code 13) wird ein Algorithmus bestehend aus 4 Abschnitten (plus Stop) und 4 Takten pro Abschnitt gezeigt. Es handelt sich um ein Programm, das die Summe der ersten 4 Quadratzahlen (0+1+4+9=14) berechnet und die sog. Abschnitts- und Taktsteuerung demonstriert. Die Abschnittssteuerung erfolgt in einem endlichen Automaten. Jeder Abschnitt verwaltet die Zahl seiner Takte, die er benötigt, selbst. Alle Abschnitte werden aus Gründen der Vereinfachung mit Hilfe derselben Prozedur implementiert. Das Programm kann leicht auf die Summe der ersten 8 Quadratzahlen bzw. auf 8 Abschnitte erweitert werden.

```
-- Package fuer Konstanten und Typdefinitionen:
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package Konstanten is
-- Konstantendeklarationen
constant HoechsterWertInAbschnitte: integer:=7;
-- mache im Abschnittsautomaten 8 Abschnitte, gezaehlt von 0 bis 7
constant GroesseVonAbschnitte: integer:=HoechsterWertInAbschnitte+2;
-- Aufgrund des Abschnitts "Ende" und aufgrund der Zaehlung von 0 an gibt
-- es 2 Abschnitte mehr als HoechsterWertInAbschnitte gross ist
constant Abschnitt0: integer := 0;
-- verwende symbolische Namen fuer die Abschnitte anstelle von Zahlen
constant Abschnitt1: integer := 1;
constant Abschnitt2: integer := 2;
constant Abschnitt3: integer := 3;
constant Abschnitt4: integer := 4;
constant Abschnitt5: integer := 5;
constant Abschnitt6: integer := 6;
constant Abschnitt7: integer := 7;
constant Ende: integer := 8;

constant GroesseVonGanzeZahlen: integer:=8;
-- Verwende hoechstens die ersten 8 ganzen Zahlen von 0 bis 7
-- => 3 Bit-Groesse
constant HoechsterWertInGanzeZahlen: integer := GroesseVonGanzeZahlen-1;
-- Die ganzen Zahlen beginnen mit 0, deshalb ist HoechsterWertInGanzeZahlen
-- um 1 kleiner
constant HoechsterWertVonSumme: integer:=
HoechsterWertInGanzeZahlen*HoechsterWertInGanzeZahlen*HoechsterWertInGanzeZahlen;
-- Abschaetzung fuer die groesstmoegliche Summe. Der Wert hat die
-- Komplexitaet O(N**3).
-- Der echte Wert der Summe der ersten n Quadratzahlen ist (1/6)n(n+1)(2n+1)
-- => HoechsterWertVonSumme hat mehr Bits als HoechsterWertInGanzeZahlen
constant HoechsterWertVonTaktzaehler: integer:=GroesseVonGanzeZahlen;
-- mache nicht mehr Takte als es ganze Zahlen im Feld gibt
constant HoechsterWertAmAusgabeport: integer:= HoechsterWertVonSumme;
-- es kann hoechstens der HoechsterWertVonSumme ausgegeben werden
-- Typdeklarationen
type TypAbschnitte is array(0 to GroesseVonAbschnitte-1) of integer
  range 0 to HoechsterWertInAbschnitte+1;
-- Programm hat GroesseVonAbschnitte an Abschnitten
type TypGanzeZahlen is array(0 to GroesseVonGanzeZahlen-1) of integer
  range 0 to HoechsterWertInGanzeZahlen;
constant GanzeZahlen : TypGanzeZahlen := (0,1,2,3,4,5,6,7);
-- synth. ROM. GanzeZahlen enthaelt die ersten ganzen Zahlen bis
-- HoechsterWertInGanzeZahlen.
-- Initialisiere mit 0,1,2,3,...,HoechsterWertInGanzeZahlen
end Konstanten;

--Package fuer Prozeduren:
library IEEE;
use IEEE.STD_lOGIC_1164.all;
use IEEE.STD_lOGIC_ARITH.All;
use IEEE.STD_lOGIC_UNSIGNED.All;
use STD.TEXTIO.All;
use work.PCK_FIO.all; -- print package
use work.Konstanten.all;

package Prozeduren is
file Bildschirm : text open write_mode is "STD_OUTPUT";
```

```
-- Bildschirmausgabe bei Modelsim

procedure Abschnittsprozedur(
  signal i: inout integer range 0 to HoechsterWertVonTaktzaehler;
  -- i = Taktzaehler
  signal j: inout integer range 0 to HoechsterWertInAbschnitte+1;
  -- j = Abschnittszaehler
  -- variable SMax: integer:= Summationsobergrenze;
  variable SMax: integer range 0 to 4;
  -- SMax = sagt, wie weit summiert werden soll
  Signal S: inout integer range 0 to HoechsterWertVonSumme;
  -- S enthaelt die Summe der Quadratzahlen
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport);
end Prozeduren;

package body Prozeduren is

procedure Abschnittsprozedur(signal i: inout integer
  range 0 to HoechsterWertVonTaktzaehler;
  signal j: inout integer range 0 to HoechsterWertInAbschnitte+1;
  variable SMax: integer range 0 to 4;
  signal S: inout integer range 0 to HoechsterWertVonSumme;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport) is
  -- Deklarationen
  type TypVonH is array(0 to GroesseVonGanzeZahlen-1) of integer
    range 0 to HoechsterWertVonSumme;
  -- Typdeklaration von Hilfsgroessen dort, wo sie direkt benoetigt wird.
  variable H: TypVonH := (others=>0);
  -- H dient als Hilfsgroesse fuer GroesseVonGanzeZahlen an Zwischenwerten.
  -- Initialisierung mit 0.
  variable BildschirmZeile: line;
  -- noetig fuer die Bildschirmausgabe bei Modelsim
begin

if i<SMax then
  -- Summiere die ersten Quadratzahlen von 0 bis zur Summationsobergrenze
  H(i) := GanzeZahlen(i)*GanzeZahlen(i);
  -- synth. Multiplizierer für die ganzen Zahlen
  -- Schreiben von H vor dem Lesen => kein Register
  -- => H(i) erhalt seinen Wert sofort zugewiesen.
  -- H(i) wird vor dem naechsten Takt fertig.
  S<= S + H(i);
  -- synth. Akkumulator zum Aufaddieren der Quadratzahlen.
  -- Wert wird einen Takt spaeter gueltig
  i <= i+1; -- synth. Taktzaehler. Wert wird einen Takt spaeter gueltig.
elsif i= SMax then
  -- Summationsobergrenze ist erreicht
  Ausgabe <= S;
  -- Gebe am Schluss der Prozedur den letzten Wert von S aus.
  -- Register. Wert wird einen Takt spaeter gueltig.
  fprint(Bildschirm,BildschirmZeile,"Abschnitt1, S= %d\n", fo(S));
  -- Gebe in Modelsim die erreichte Summe auch auf dem Bildschirm aus
  -- Prozedur. Wert wird einen Takt spaeter gueltig.
  -- Bereinige Taktzaehler und erhoehe Abschnittszaehler
  if j < HoechsterWertInAbschnitte then
    j <= j+1;
    -- Gehe zum naechsten Abschnitt oder zu Ende
    -- Synth. Abschnittszaehler. Wert wird einen Takt spaeter gueltig.
    i <= 0;
    -- Ruecksetzen des Taktzaehlers zur Wiederverwendung des Taktzaehlers.
```

```
    -- Wert wird einen Takt spaeter gueltig.
    S <= 0;
    -- Ruecksetzen der Summe zur Wiederverwendung des Signals.
    -- Wert wird einen Takt spaeter gueltig.
  else -- j >= HoechsterWertInAbschnitte. Dies ist ein Fehler.
    j <= Ende; -- stop
  end if; -- i<HoechsterWertInAbschnitte
end if; -- i<SMax
end Abschnittsprozedur;
end Prozeduren;


--Entity, architecture und process
library IEEE;
use IEEE.STD_lOGIC_1164.All;
use work.Prozeduren.all;
use work.Konstanten.all;
entity Algorithmus1 is

Port (clk : in STD_lOGIC;
      NotwendigerAusgabeport : inout integer
      range 0 to HoechsterWertAmAusgabeport);
end Algorithmus1;

architecture One of Algorithmus1 is
  -- Signaldeklarationen
  -- starte mit Abschnitt0
  signal Abschnittszaehler:  integer
    range 0 to HoechsterWertInAbschnitte+1:= 0;
  -- Starte mit Takt 0.
  signal Taktzaehler:  integer range 0 to HoechsterWertVonTaktzaehler := 0;
  -- Maximum ist HoechsterWertVonTaktzaehler an Takten pro Abschnitt
  signal SummeQuadratzahlen:  integer
    range 0 to HoechsterWertVonSumme := 0;
  -- Starte mit einer Summe der Quadratzahlen von 0

begin

process(clk)
  variable Summationsobergrenze: integer:=4;
  -- Berechne die Summe der ersten 4 Quadratzahlen
  -- => Summationsobergrenze ist 3 Bit-Groesse
  -- Summationsobergrenze darf nicht groesser als GroesseVonGanzeZahlen sein
begin
if clk'event and clk = '1' then
  if Abschnittszaehler = Abschnitt0 then
    Abschnittsprozedur(Taktzaehler, Abschnittszaehler,
Summationsobergrenze, SummeQuadratzahlen, NotwendigerAusgabeport);
  elsif Abschnittszaehler = Abschnitt1 then
    Abschnittsprozedur(Taktzaehler, Abschnittszaehler,
Summationsobergrenze, SummeQuadratzahlen, NotwendigerAusgabeport);
  elsif Abschnittszaehler = Abschnitt2 then
    Abschnittsprozedur(Taktzaehler, Abschnittszaehler,
Summationsobergrenze, SummeQuadratzahlen, NotwendigerAusgabeport);
  elsif Abschnittszaehler = Abschnitt3 then
    Abschnittsprozedur(Taktzaehler, Abschnittszaehler,
Summationsobergrenze, SummeQuadratzahlen, NotwendigerAusgabeport);
  elsif Abschnittszaehler = Ende then
    NULL; -- Mache nichts
  end if;
```

```
end if; -- clk'event and clk = '1'
end process;
end One;
```

Programmcode 13: Algorithmus bestehend aus 4 Abschnitten (plus Stop) und 4 Takten pro Abschnitt.

9.2 Gekoppelte Programmabschnitte mit Parameteraustausch

Im nächsten Programmierbeispiel werden unterschiedliche Prozeduren aufgerufen, die über Parameterübergaben gekoppelt sind. Dies ist in den meisten Anwendungen der Standardfall.

```
--Zeigt, wie man einen Algorithmus, der aus mehreren unterschiedlichen
--Teilen (Abschnitt0 und Abschnitt1) besteht, synthetisiert
--und wie die Parameterübergabe zwischen den Abschnitten ablaeuft. Die
--Abschnittssteuerung erfolgt in einem endlichen Automaten. Jeder
--Abschnitt wird in einer Prozedur berechnet, die Teil eines
--Programmpakets ist.
--Für die verwendeten Felder wird eine konfigurierbare Feldgrenzenverwaltung
--eingesetzt, da der Output einer Prodezur den Input der nachfolgenden
--bildet. In Abschnitt0 wird die Summe der ersten 4 ganzen Zahlen d.h.
--0+1+2+3=6 berechnet.
--In Abschnitt1 wird anhand des Ergebniswertes aus Abschnitt0
--die Summe der ersten 6 Quadratzahlen (0+1+4+9+16+25=55) bestimmt.


--Package fuer Konstanten und Typdefinition:
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package Konstanten is

constant GroesseVonAbschnitte: integer:=3;
-- Programm hat 2 Abschnitte + Stop
type TypAbschnitte is (Abschnitt0, Abschnitt1, Ende);
constant GroesseVonGanzeZahlen: integer:=8;
-- Verwende hoechstens die ersten 8 ganzen Zahlen von 0 bis 7
constant HoechsterWertInGanzeZahlen: integer := GroesseVonGanzeZahlen-1;
-- Die ganzen Zahlen beginnen mit 0, deshalb ist
-- HoechsterWertInGanzeZahlen um 1 kleiner
type TypGanzeZahlen is array(0 to GroesseVonGanzeZahlen-1) of integer
range 0 to HoechsterWertInGanzeZahlen ;
constant GanzeZahlen : TypGanzeZahlen:= (0,1,2,3,4,5,6,7);
-- synth. ROM. Geltungsbereich der Konstanten ist im ganzen Programm
-- Enthaelt die ersten ganzen Zahlen bis HoechsterWertInGanzeZahlen.
-- Initialisiere mit 0,1,2,…,HoechsterWertInGanzeZahlen
-- Deklarationen fuer Abschnitt0
constant Abschnitt0ZahlDerSummanden: integer:=4;
-- Berechne die Summe der ersten 4 Ganzzahlen
-- Abschnitt0ZahlDerSummanden darf nicht groesser als GroesseVonGanzeZahlen sein
-- Hier kann man waehlen, wie viele erste ganze Zahlen addiert werden sollen
constant Abschnitt0HoechsterWertVonSumme: integer:=
HoechsterWertInGanzeZahlen*(HoechsterWertInGanzeZahlen+1);
-- Abschaetzung fuer die groesstmoegliche Summe. Der Wert hat die
-- Komplexitaet O(N**2).
-- Der echte Wert der Summe der ersten n Zahlen ist(1/2)n(n+1)
-- Deklarationen fuer Abschnitt1
constant Abschnitt1HoechsterWertVonQuadratsumme: integer:=
HoechsterWertInGanzeZahlen*(HoechsterWertInGanzeZahlen+1)*
```

```
(HoechsterWertInGanzeZahlen+1);
-- Abschaetzung fuer die groesstmoegliche Summe. Der Wert hat die
-- Komplexitaet O(N**3).
-- Der echte Wert der Summe der ersten n Quadratzahlen ist (1/6)n(n+1)(2n+1)
-- Allgemeine Deklarationen
constant HoechsterWertVonTaktzaehler:
integer:=GroesseVonGanzeZahlen;
-- Abschaetzung fuer den groessten Wert des Taktzaehlers
constant HoechsterWertAmAusgabeport: integer:=
Abschnitt1HoechsterWertVonQuadratsumme;
-- es kann hoechstens Abschnitt1HoechsterWertVonQuadratsumme ausgegeben werden

end Konstanten;

--Package fuer Prozeduren:
library IEEE;
use IEEE.STD_lOGIC_1164.all;
use IEEE.STD_lOGIC_ARITH.All;
use IEEE.STD_lOGIC_UNSIGNED.All;
use STD.TEXTIO.All;
use work.PCK_FIO.all; -- print package
use work.Konstanten.all;

package Prozeduren is
file Bildschirm : text open write_mode is "STD_OUTPUT";
-- Bildschirmausgabe bei Modelsim

procedure ProzedurAbschnitt0(
  signal Index: inout integer range 0 to HoechsterWertVonTaktzaehler;
  signal Abschnitt: inout TypAbschnitte;
  signal Summe: inout integer range 0 to Abschnitt0HoechsterWertVonSumme;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport);

procedure ProzedurAbschnitt1(
  signal Index: inout integer range 0 to HoechsterWertVonTaktzaehler;
  signal Abschnitt: inout TypAbschnitte;
  signal AnzahlSummanden: in integer range 0 to
    Abschnitt0HoechsterWertVonSumme;
  signal Quadratsumme: inout integer range 0 to
    Abschnitt1HoechsterWertVonQuadratsumme;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport);

end Prozeduren;

package body Prozeduren is
-------------------------------------------------------------------------

procedure ProzedurAbschnitt0(
  signal Index: inout integer range 0 to HoechsterWertVonTaktzaehler;
  signal Abschnitt: inout TypAbschnitte;
  signal Summe: inout integer range 0 to Abschnitt0HoechsterWertVonSumme;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport) is
  variable BildschirmZeile: line;
  -- Fuer Modelsim
begin
if Abschnitt0ZahlDerSummanden>GroesseVonGanzeZahlen then
  fprint(Bildschirm, BildschirmZeile,
  "Abschnitt0ZahlDerSummanden>GroesseVonGanzeZahlen:
Abschnitt0ZahlDerSummanden= %d\n",
  fo(Abschnitt0ZahlDerSummanden));
```

```
  Abschnitt <= Ende; -- Fehler
end if; -- AnzahlSummanden>GroesseVonGanzeZahlen

-- ZahlDerSummanden passt
if Index<=Abschnitt0ZahlDerSummanden-1 then
  -- Beispiel: Wenn Abschnitt0ZahlDerSummanden=4 ist, dann laeuft der Index
  -- von 0 bis 3
  Summe <= Summe + GanzeZahlen(Index);
  -- synth. Akkumulator
  -- Wert wird einen Takt spaeter gueltig.
  -- Summiere die ganzen Zahlen auf
  -- GanzeZahlen ist ein Feld, das im Konstanten-Paket deklariert wurde.
  -- Es gilt als globale Konstante, deren Geltungsbereich ueberall ist.
  Index <= Index+1;
  -- synth. Taktzaehler.
  -- Wert wird einen Takt spaeter gueltig.
else
  -- letzte Summation wurde ausgefuehrt
  Ausgabe <= Summe;
  -- Letzten Wert von Summe ausgeben.
  -- Register. Wert wird einen Takt spaeter gueltig.
  fprint(Bildschirm, BildschirmZeile,"Abschnitt0, Summe= %d\n", fo(Summe));
  -- Gebe bei Modelsim Summe auch auf dem Bildschirm aus
  Abschnitt <= Abschnitt1;
  -- Gehe zum naechsten Abschnitt.
  -- Wert wird einen Takt spaeter gueltig.
  Index <= 0;
  -- Wiederverwendung des Taktzaehlers.
  -- Wert wird einen Takt spaeter gueltig.
end if; -- Index<=Abschnitt0ZahlDerSummanden-1
end ProzedurAbschnitt0;
--------------------------------------------------------------------------

procedure ProzedurAbschnitt1(
  signal Index: inout integer range 0 to HoechsterWertVonTaktzaehler;
  signal Abschnitt: inout TypAbschnitte;
  signal AnzahlSummanden: in integer range 0 to Abschnitt0HoechsterWertVonSumme;
  signal Quadratsumme: inout integer range 0 to
    Abschnitt1HoechsterWertVonQuadratsumme;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport)is

  variable Quadrat: integer range 0 to Abschnitt1HoechsterWertVonQuadratsumme :=0;
  -- Fuer Zwischenwerte
  variable BildschirmZeile: line;
  -- Fuer Modelsim
begin
if AnzahlSummanden>GroesseVonGanzeZahlen then
  fprint(Bildschirm, BildschirmZeile,
  "AnzahlSummanden>GroesseVonGanzeZahlen: AnzahlSummanden= %d\n",
  fo(AnzahlSummanden));
  Abschnitt <= Ende; -- Fehler
end if; -- AnzahlSummanden>GroesseVonGanzeZahlen

-- AnzahlSummanden passt
if Index<=AnzahlSummanden-1 then
  -- Beispiel: Wenn AnzahlSummanden=4 ist, dann laeuft der Index
  -- von 0 bis 3
  Quadrat := GanzeZahlen(Index)*GanzeZahlen(Index);
  -- synth. 1 Multiplizierer
  -- Schreiben vor dem Lesen. Kein Register. Wert wird sofort gueltig.
```

```
  Quadratsumme<= Quadratsumme + Quadrat;
  -- synth. Akkumulator.
  -- Register, da Signal. Wert wird einen Takt spaeter gueltig.
  Index <= Index+1; -- synth. Zaehler.
  -- Wert wird einen Takt spaeter gueltig.
else -- Index > AnzahlSummanden-1
  -- Alle Quadratsummen wurden aufaddiert
  Ausgabe <= Quadratsumme;
  -- letzter Wert von Quadratsumme wird ausgegegeben.
  -- Wert wird einen Takt spaeter gueltig.
  fprint(Bildschirm, BildschirmZeile,"Abschnitt1, Quadratsumme= %d\n",
  fo(Quadratsumme));
  -- Gebe bei Modelsim die Quadratsumme auf dem Bildschirm aus
  -- Abschnitt <= Ende; -- Gehe zum naechsten Abschnitt.
  Abschnitt <= Ende; -- Gehe zum naechsten Abschnitt
  --Wert wird einen Takt spaeter gueltig.
end if; -- Index<AnzahlSummanden-1
end ProzedurAbschnitt1;
--------------------------------------------------------------------------
end Prozeduren;

-- Hauptprogramm
library IEEE;
use IEEE.STD_lOGIC_1164.All;
use work.Konstanten.all;
use work.Prozeduren.all;

entity Algorithmus2 is
Port (clk: in STD_lOGIC;
      NotwendigerAusgabeport: inout integer range 0 to
      HoechsterWertAmAusgabeport);
end Algorithmus2;

architecture One of Algorithmus2 is
-- Deklarationen der Signale fuer Abschnitt 0
signal Taktzaehler: integer range 0 to HoechsterWertVonTaktzaehler := 0;
-- Starte mit Takt und Index 0.
signal Abschnittszaehler: TypAbschnitte:= Abschnitt0;
-- starte mit Abschnitt0
-- Maximum ist HoechsterWertVonTaktzaehler an Takten pro Abschnitt
signal SummeAbschnitt0: integer range 0 to Abschnitt0HoechsterWertVonSumme:= 0;
-- Starte mit der Summe 0
-- Deklarationen der Signale fuer Abschnitt 1
signal Quadratsumme: integer range 0 to
Abschnitt1HoechsterWertVonQuadratsumme :=0;
-- Starte mit der Quadratsumme 0
begin

process(clk)
  begin
    if clk'event and clk = '1' then
      if Abschnittszaehler = Abschnitt0 then
        ProzedurAbschnitt0(Taktzaehler, Abschnittszaehler, SummeAbschnitt0,
                           NotwendigerAusgabeport);
      elsif Abschnittszaehler = Abschnitt1 then
        ProzedurAbschnitt1(Taktzaehler, Abschnittszaehler, SummeAbschnitt0,
                           Quadratsumme, NotwendigerAusgabeport);
      elsif Abschnittszaehler = Ende then
        NULL; -- Mache nichts
      end if; -- Abschnittszaehler = Abschnitt0
```

```
    end if; -- clk'event and clk = '1'
  end process;
end One;
```

Programmcode 14: Zwei unterschiedliche Prozeduren, die über Parameterübergaben gekoppelt sind.

9.3 Gleichungslöser für eine Wärmeleitungsgleichung Version 1

Hier wird anhand der Berechnung der Wärmeausbreitung in einer Platte gezeigt, wie man komplexe VHDL-Programme erfolgreich synthetisiert. Das Beispielprogramm ist mit Absicht rel. umfangreich und dadurch unübersichtlich. Es soll zeigen, dass die Verwendung von Moduln und Unterprogrammen für gute Übersichtlichkeit notwendig ist.

```
-- Zeigt, wie man einen komplexen Algorithmus implementiert.
-- Auch hier kann das Programm nach jedem Neustart an die
-- Stelle zurückfinden, an der es beim vorangegangenen Aufruf stand. Dafür
-- sorgen vier Zustandsgrößen: Abschnittszaehler, Iterationsindex,
-- Zeilenindex und Spaltenindex sowie viele Fallunterscheidungen. Die
-- Indizes werden als Signal deklariert, um eine Synthese als Zähler zu
-- garantieren. Das Problem bei diesem Beispiel ist die geringe
-- Uebersichtlichkeit der Software, die dadurch entsteht, dass keine
-- mehrfachen Unterprogrammaufrufe eingesetzt werden.
-- Es wird ein Beispiel aus der Physik berechnet, und zwar eine einfache
-- Waermeleitungsgleichung für eine ebenen Platte mit einer
-- Waermeeinspeisung (=Wärmequelle) am linken Rand.

--Package fuer Konstanten und Typdefinition:
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package Konstanten is
 type Abschnitte is (Abschnitt0, Ende); -- enumeration type
 -- Programm hat nur einen Abschnitt + Ende
 constant RandTemperatur: integer:=127;
 -- Legt die Temperatur am linken Rand der Platte zu Beginn der Iteration fest.
 -- Alle anderen Raender haben Null Grad (= keine Waermeeinspeisung)
 constant LaengeDerPlatte: integer:=16;
 -- Platte habe die Laenge 16.
 -- Platte sei rechteckig
 constant BreiteDerPlatte: integer:=8;
 -- Platte habe die Breite 8.
 constant HoechsterWertInLaenge : integer:= LaengeDerPlatte-1;
 -- Zeilenindex zur Adressierung der Platte laeuft von 0 bis LaengeDerPlatte-1
 constant HoechsterWertInBreite : integer:= BreiteDerPlatte-1;
 -- Spaltenindex zur Adressierung der Platte laeuft von 0 bis BreiteDerPlatte-1
  type TypPlatte is array (0 to HoechsterWertInLaenge, 0 to
  HoechsterWertInBreite) of integer range 0 to RandTemperatur;
  -- Platte ist ein 2D-Feld, mit Elementen, die maximal so warm wie die
  -- Einspeisetemperatur werden koennen.
  constant HoechsterWertVonMittelwert: integer:=RandTemperatur;
  -- Legt das hoechste Ergebnis der Mittelwertbildung fest. Platte kann nie heisser
als der Rand werden.
  constant HoechsterWertIterationszaehler: integer:=127;
  -- Iteriere 128 Mal ueber die Waermegleichung. Beende danach das Programm.
  constant HoechsterWertAmAusgabeport: integer:= RandTemperatur;
  -- Es kann hoechstens der Wert der RandTemperatur ausgegeben werden
end Konstanten;
```

```
--Package fuer Prozeduren:
library IEEE;
use IEEE.STD_lOGIC_1164.all;
use IEEE.STD_lOGIC_ARITH.All;
use IEEE.STD_lOGIC_UNSIGNED.All;
use work.Konstanten.all;
use STD.TEXTIO.All;
use work.PCK_FIO.all; -- print package

package Prozeduren is
  file Bildschirm : text open write_mode is "STD_OUTPUT";
  -- Bildschirmausgabe bei Modelsim
  --variable Fall: inout integer range 1 to 9; -- nur fuer Debug

  procedure debug(
    variable Fall: inout integer range 1 to 9;
    -- gibt an, um welchen Fall es sich handelt
    signal Z: inout integer range 0 to HoechsterWertInLaenge;
    -- gibt die Zeile eines Plattenelements aus
    signal S: inout integer range 0 to HoechsterWertInBreite;
    -- gibt die Spalte eines Plattenelements aus
    variable Mittel: inout integer range 0 to HoechsterWertVonMittelwert);
    -- gibt den Wert nach der Mittelung eines Elements aus;

 procedure ProzedurWaermegleichung(
    variable Fall: inout integer range 1 to 9;
    -- gibt an, um welchen Fall es sich handelt
    signal Abschnitt: inout Abschnitte;
    -- Unterscheidet zwischen Prozeduraufruf und Ende
    signal Zeile: inout integer range 0 to HoechsterWertInLaenge;
    -- Adressiert die Zeile eines Plattenelements
    signal Spalte: inout integer range 0 to HoechsterWertInBreite;
    -- Adressiert die Spalte eines Plattenelements
    variable Platte: inout TypPlatte;
    -- 2D-Feld, uber das iteriert wird. Muss als Register synth. werden.
    variable PlatteNeu: inout TypPlatte;
    -- Wert der Platte nach der Mittelung eines Elements.
    -- Muss als Register synth. werden.
    signal Iterationen: inout integer range 0 to HoechsterWertIterationszaehler;
    -- Zaehlt die Zahl der Iterationen
    signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport);
    -- notwendiger Ausgabeport;
end Prozeduren;

package body Prozeduren is
  procedure debug(
    variable Fall: inout integer range 1 to 9;
    signal Z: inout integer range 0 to HoechsterWertInLaenge;
    -- gibt die Zeile eines Plattenelements aus
    signal S: inout integer range 0 to HoechsterWertInBreite;
    -- gibt die Spalte eines Plattenelements aus
    variable Mittel: inout integer range 0 to HoechsterWertVonMittelwert)is
    -- gibt den Wert nach der Mittelung eines Elements aus
    variable BildschirmZeile: line;
    -- Fuer Modelsim
    begin
      fprint(Bildschirm, BildschirmZeile, "Fall=%d\n", fo(Fall));
      fprint(Bildschirm, BildschirmZeile, "Z=%d\n", fo(Z));
      fprint(Bildschirm, BildschirmZeile, "S=%d\n", fo(S));
      fprint(Bildschirm, BildschirmZeile, "Mittel=%d\n", fo(Mittel));
```

```
end debug;

procedure ProzedurWaermegleichung(
  variable Fall: inout integer range 1 to 9;
  -- gibt an, um welchen Fall es sich handelt
  signal Abschnitt: inout Abschnitte;
  signal Zeile: inout integer range 0 to HoechsterWertInLaenge;
  signal Spalte: inout integer range 0 to HoechsterWertInBreite;
  variable Platte:inout TypPlatte;
  variable PlatteNeu: inout TypPlatte;
  signal Iterationen: inout integer range 0 to HoechsterWertIterationszaehler;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport) is
  variable BildschirmZeile: line;
  -- Fuer Modelsim
  variable Mittelwert: integer range 0 to HoechsterWertVonMittelwert:=0;
  -- Enthaelt die Mittelung ueber die Nachbarpunkte in Nord, Sued, West und Ost
  begin
    -- Mache alle gewunschten Interationen
    -- Strukturiere jede Iteration
    if Iterationen<=HoechsterWertIterationszaehler then
     -- Maximale Zahl der Iterationen ist noch nicht ueberschritten
     -- Mache eine Iteration
     -- Behandle sowohl die innere Platte als auch die Rander
     -- Behandle die 4 Eckpunkte und die 4 Rander extra
     -- Behandle alle 9 Falle von Mittelwertbildungen uber die Nachbarpunkte
     -- Der Zeilenindex soll langsamer als der Spaltenindex laufen
     if Zeile=0 and Spalte=0 then
       -- Fall 1: Ecke oben links. Beginne hier
       Mittelwert := (Platte(Zeile+1,Spalte)+ -- Sued
                      Platte(Zeile,Spalte+1))/2; --Ost
       -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in Sued und Ost.
       -- Mittelwert ist kein Register. Wird sofort gueltig.
       -- Von Platte wird zuerst gelesen => Register.
       Fall:= 1;
       debug(Fall, Zeile, Spalte, Mittelwert); --loeschen?
       -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
       PlatteNeu(Zeile,Spalte) := Mittelwert;
       -- Register, da Zuweisung nur im if-Zweig, aber nicht im else-Zweig.
       -- Wird einen Takt spaeter gueltig.
       -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
       Ausgabe <= Mittelwert;
       -- Register. Wird einen Takt spaeter gueltig.
       Spalte <= Spalte+1; -- gehe durch die erste Zeile
       -- Synth. Spaltenzaehler
       -- Wird einen Takt spaeter gueltig.
     elsif Zeile=0 and Spalte=HoechsterWertInBreite then
       -- Fall 2: Ecke oben rechts
       Mittelwert := (Platte(Zeile,Spalte-1)+ -- West
                      Platte(Zeile+1,Spalte))/2; -- Sued
       -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in Sued und West.
       Fall:= 2;
       debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
       -- Bringe am Ende einer Zeile den Zeilenzaehler auf den neuesten Stand
       -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
       PlatteNeu(Zeile,Spalte) := Mittelwert;
       -- Register, da Zuweisung nur im if-Zweig, aber nicht im else-Zweig.
       -- Wird einen Takt spaeter gueltig.
       -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
       Ausgabe <= Mittelwert;
       -- Register. Wird einen Takt spaeter gueltig.
```

```
  Zeile <= Zeile +1; -- gehe zur nachsten Zeile
  -- Synth. Zeilenzaehler
  -- Wird einen Takt spaeter gueltig.
  Spalte <= 0; -- beginne wieder mit der ersten Spalte
  -- Wird einen Takt spaeter gueltig.
elsif Zeile=HoechsterWertInLaenge and Spalte=0 then
  -- Fall 3: Ecke unten links
  Mittelwert := (Platte(Zeile,Spalte+1)+ -- Ost
                 Platte(Zeile-1,Spalte))/2; -- Nord
  -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in Nord und Ost.
  Fall:= 3;
  debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
  -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
  PlatteNeu(Zeile,Spalte) := Mittelwert;
  -- Register, da Zuweisung nur im if-Zweig, aber nicht im else-Zweig.
  -- Wird einen Takt spaeter gueltig.
  -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
  Ausgabe <= Mittelwert;
  -- Register. Wird einen Takt spaeter gueltig.
  Spalte <= Spalte+1; -- gehe durch die letzte Zeile
  -- Wird einen Takt spaeter gueltig.
elsif
 Zeile=HoechsterWertInLaenge and Spalte=HoechsterWertInBreite then
  -- Fall 4: Ecke unten rechts
  Mittelwert := (Platte(Zeile,Spalte-1)+ -- West
                 Platte(Zeile-1,Spalte))/2; -- Nord
  -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in Nord und West.
  Fall:= 4;
  debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
  -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
  PlatteNeu(Zeile,Spalte) := Mittelwert;
  -- Register, da Zuweisung nur im if-Zweig, aber nicht im else-Zweig
  -- Wird einen Takt spaeter gueltig. Der letzte Wert von PlatteNeu,
  -- d.h. PlatteNeu(HoechsterWertInLaenge,HoechsterWertInBreite)
  -- steht leider erst einen Takt spaeter zur Verfugung. Der Fall 4
  -- wird dann aber nicht mehr aufgerufen.
  -- D.h. PlatteNeu(HoechsterWertInLaenge,HoechsterWertInBreite)
  -- muss beim Umkopieren auf Platte separat behandelt werden.
  -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
  Ausgabe <= Mittelwert;
  -- Register. Wird einen Takt spaeter gueltig.
  -- Letzte Zeile und letzte Spalte sind erreicht
  -- Eine Iteration ist durchgelaufen
  -- Mache Aufraeumarbeiten am Ende jeder Iteration
  -- Dieser Fall tritt nur einmal alle
  -- LaengeDerPlatte*BreiteDerPlatte Aufrufe auf.
  -- Kopiere nach jeder Iteration PlatteNeu nach Platte, um Platte
  -- auf den neuesten Stand zu bringen.
  Platte := PlatteNeu; -- Kopiert fast das ganze Feld in einem Befeh
  -- Der letzte Wert von Platte kann leider mit diesem Befehl nicht
  -- kopiert werden, da er erst im nachsten Takt zur Verfugung steht
  -- Er muss separat kopiert werden.
  -- Beim Ubersetzen wird ein Warning fuer
  -- PlatteNeu(HoechsterWertInLaenge,HoechsterWertInBreite)
  -- auftreten, da dieses Feldelement nicht gelesen wird!
  Platte(Zeile,Spalte) := Mittelwert;
  -- Mittelwert ist kein Register. Der Wert wird sofort gueltig.
  -- Der letzte Wert von PlatteNeu wird auf diese Art kopiert.
  -- Wert von Platte wird einen Takt spaeter gueltig.
  -- Beginne wieder mit der ersten Zeile und der ersten Spalte fuer
```

```
  -- die naechste Iteration.
  Zeile <= 0;
  Spalte <= 0;
  -- Wird einen Takt spaeter gueltig.
  -- Bringe am Ende jeder Iteration den Iterationszaehler auf den
  -- neuesten Stand
  if Iterationen<HoechsterWertIterationszaehler then
    Iterationen <= Iterationen+1; -- synth. IterationsZaehler.
    -- Wert wird einen Takt spaeter gueltig.
  else -- Iterationen=HoechsterWertIterationszaehler
    -- Iterationen hat den HoechsterWertIterationszaehler erreicht.
    -- Erhoehe Iterationen nicht weiter.
    -- Beende Programm
    Abschnitt <= Ende;
    -- Wird einen Takt spaeter gueltig.
    -- Programmende erreicht
  end if; -- Iterationen<HoechsterWertIterationszaehler
   -- Gebe bei Modelsim die Platte am Anfang und nach jeder 8.
   -- Iteration, sowie am Ende der Erhoehungen des Iterationszaehlers
   -- auf dem Bildschirm aus.
  if Iterationen=0 or (Iterationen mod 7) = 0 or
    Iterationen=HoechsterWertIterationszaehler-1
    -- Wert von Iterationen ist an dieser Stelle immer um eins
    -- kleiner, da die Zuweisung noch nicht wirksam wurde.
    then
    fprint(Bildschirm, BildschirmZeile, "Iterationen=%d\n",
      fo(Iterationen));
    fprint(Bildschirm, BildschirmZeile, "Platte:\n");
    for i in 0 to HoechsterWertInLaenge loop -- Gehe uber alle Zeilen
      -- benuetze eigene Indizes fuer die Schleifen
      for j in 0 to HoechsterWertInBreite loop
      -- Gehe uber alle Spalten
        fprint(Bildschirm, BildschirmZeile," %d", fo(Platte(i,j)));
      end loop; -- j
      fprint(Bildschirm, BildschirmZeile, "\n");
    end loop; -- i
  else -- (Iterationen mod 8) /= 0
    null; -- gebe bei allen anderen Iterationen nichts aus
  end if; -- (Iterationen mod 8) = 0
elsif Zeile=0 and 0<Spalte and Spalte<HoechsterWertInBreite then
   -- Fall 5: oberer Rand ohne Ecken
   -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in
   -- Sued, Ost und West. Da nicht durch 3 dividiert werden kann,
   -- wird nur West und Ost verwendet und naherungsweise durch 2
   -- dividiert
   Mittelwert := (Platte(Zeile,Spalte-1)+ -- West
                  Platte(Zeile,Spalte+1))/2; -- Ost
   Fall:= 5;
   debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
   -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
   PlatteNeu(Zeile,Spalte) := Mittelwert;
   -- Register, da Zuweisung nur im if-Zweig, aber nicht im
   -- else-Zweig.
   -- Wird einen Takt spaeter gueltig.
   -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
   Ausgabe <= Mittelwert;
   -- Register. Wird einen Takt spaeter gueltig.
   Spalte <=Spalte+1; -- bleibe in der ersten Zeile
   -- Wird einen Takt spaeter gueltig.
 elsif Zeile=HoechsterWertInLaenge and 0<Spalte and
```

```
  Spalte<HoechsterWertInBreite then
  -- Fall 6: unterer Rand ohne Ecken
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in
  -- Nord, Ost und West. Da nicht durch 3 dividiert werden kann,
  -- wird nur West und Ost verwendet und naherungsweise durch 2
  -- dividiert
  Mittelwert := (Platte(Zeile,Spalte-1)+ -- West
                 Platte(Zeile,Spalte+1))/2; -- Ost
  Fall:= 6;
  debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
  -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
  PlatteNeu(Zeile,Spalte) := Mittelwert;
  -- Register, da Zuweisung nur im if-Zweig, aber nicht im
  -- else-Zweig.
  -- Wird einen Takt spaeter gueltig.
  -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
  Ausgabe <= Mittelwert;
  -- Register. Wird einen Takt spaeter gueltig.
   Spalte <=Spalte+1; -- bleibe in der letzten Zeile
  -- Wird einen Takt spaeter gueltig.
elsif 0<Zeile and Zeile<HoechsterWertInLaenge and Spalte=0 then
  -- Fall 7: linker Rand ohne Ecken
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn
  -- in Nord, Sued und Ost. Da nicht durch 3 dividiert werden kann,
  -- wird nur Nord und Sued verwendet und naherungsweise durch 2
  -- dividiert
  Mittelwert := (Platte(Zeile-1,Spalte)+ -- Nord
                 Platte(Zeile+1,Spalte))/2; -- Sued
  Fall:= 7;
  debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
  -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
  PlatteNeu(Zeile,Spalte) := Mittelwert;
  -- Register, da Zuweisung nur im if-Zweig, aber nicht im
  -- else-Zweig.
  -- Wird einen Takt spaeter gueltig.
  -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
  Ausgabe <= Mittelwert;
  -- Register. Wird einen Takt spaeter gueltig.
  Spalte <=Spalte+1; -- bleibe in der Zeile
  -- Wird einen Takt spaeter gueltig.
elsif 0<Zeile and Zeile<HoechsterWertInLaenge and
  Spalte=HoechsterWertInBreite then
  -- Fall 8: rechter Rand ohne Ecken
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in
  -- Nord, Sued und West. Da nicht durch 3 dividiert werden kann,
  -- wird nur Nord und Sued
  -- verwendet und naherungsweise durch 2 dividiert
  Mittelwert := (Platte(Zeile-1,Spalte)+ -- Nord
                 Platte(Zeile+1,Spalte))/2; -- Sued
  Fall:= 8;
  debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
  -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
  PlatteNeu(Zeile,Spalte) := Mittelwert;
  -- Register, da Zuweisung nur im if-Zweig, aber nicht im
  -- else-Zweig.
  -- Wird einen Takt spaeter gueltig.
  -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
  Ausgabe <= Mittelwert;
  -- Register. Wird einen Takt spaeter gueltig.
  -- Alle Spalten einer Zeile sind durchgelaufen
```

```
          Zeile <=Zeile+1; -- gehe in die nachste Zeile
          -- Wird einen Takt spaeter gueltig.
          Spalte <= 0; -- beginne wieder mit der ersten Spalte
          -- Wird einen Takt spaeter gueltig.
        elsif 0<Zeile and Zeile<HoechsterWertInLaenge and
          0<Spalte and Spalte<HoechsterWertInBreite then
          -- Fall 9: im Inneren der Platte
          Mittelwert := (Platte(Zeile-1,Spalte)+Platte(Zeile+1,Spalte)+
                         Platte(Zeile,Spalte-1)+Platte(Zeile,Spalte+1))/4;
          -- Neuer Wert ist Mittelwert seiner 4 Nachbarn Nord, Sued, West
          -- und Ost.
          Fall:= 9;
          debug(Fall, Zeile, Spalte, Mittelwert);--loeschen?
          -- Aktualisiere jetzt die Platte an der Stelle [Zeile, Spalte]
          PlatteNeu(Zeile,Spalte) := Mittelwert;
          -- Register, da Zuweisung nur im if-Zweig, aber nicht im
          -- else-Zweig.
          -- Wird einen Takt spaeter gueltig.
          -- Gebe die Platte an der Stelle [Zeile, Spalte]) aus.
          Ausgabe <= Mittelwert;
          -- Register. Wird einen Takt spaeter gueltig.
          Spalte <=Spalte+1; -- bleibe in der Zeile
          -- Wird einen Takt spaeter gueltig.
        end if; -- Zeile=0 and Spalte=0. Fall 1: Ecke oben links.
        -- alle 9 Kombinationen von Zeile und Spalte wurden berucksichtigt
      else -- Iterationen>=HoechsterWertIterationszaehler
        -- Letzte Iteration wurde durchgefuehrt. Programm wurde in Fall 4
        -- beeendet.
        null; -- mache hier nichts mehr
      end if; -- Iterationen <= HoechsterWertIterationszaehler
      end ProzedurWaermegleichung;
end Prozeduren;

-- Hauptprogramm
library IEEE;
use IEEE.STD_lOGIC_1164.All;
use work.Konstanten.all;
use work.Prozeduren.all;
use STD.TEXTIO.All;
use work.PCK_FIO.all;

entity Algorithmus3 is
  Port (clk: in STD_lOGIC;
        NotwendigerAusgabeport: inout integer range 0 to
        HoechsterWertAmAusgabeport);
end Algorithmus3;

architecture One of Algorithmus3 is
  file Bildschirm : text open write_mode is "STD_OUTPUT";
  --Deklarationen fuer Signale
  signal Abschnittszaehler: Abschnitte:= Abschnitt0;
  -- starte mit Abschnitt0
  signal Zeilenindex: integer range 0 to HoechsterWertInLaenge:=0;
  -- Zeilenindex fuer die Adressierung der Platte. Beginne mit Zeile 0.
  -- Der Rand der Platte wird extra behandelt.
  signal Spaltenindex: integer range 0 to HoechsterWertInBreite:=0;
  -- Spaltenindex fuer die Adressierung der Platte. Beginne mit Spalte 0. Der Rand
  -- der Platte wird extra behandelt.
  -- Der Spaltenindex soll sich schneller aendern als der Zeilenindex
  signal Iterationsindex: integer
```

```
   range 0 to HoechsterWertIterationszaehler :=0;
  -- starte mit Iteration 0
begin

process(clk)
  variable BildschirmZeile: line;
  variable Platte: TypPlatte:=(others=>(RandTemperatur,others=>0));
  -- Am linken Rand ist die Temperatur zu Beginn = RandTemperatur sonst 0.
  -- RandTemperatur hat Spaltenindex 0. RandTemperatur klingt ab.
  variable NeuePlatte: TypPlatte:=(others=>(others=>0));
  -- NeuePlatte=Platte nach einer Iteration. Sie hat vor der Iteration die
  -- Anfangswerte 0.
  variable Fall: integer range 1 to 9;
  -- gibt an, um welchen Fall es sich handelt
  begin
    if clk'event and clk = '1' then
      if Abschnittszaehler = Abschnitt0 then
        ProzedurWaermegleichung(Fall, Abschnittszaehler, Zeilenindex,
        Spaltenindex, Platte, NeuePlatte, Iterationsindex,
        NotwendigerAusgabeport);
      else -- Abschnittszaehler = Ende
        -- Programmende erreicht
        -- Hier kann nichts mehr gemacht werden
        null;
      end if; -- Abschnittszaehler = Abschnitt0
    end if; -- clk'event and clk = '1'
end process;
end One;
```

Programmcode 15: Komplexer Algorithmus in VHDL (Wärmeleitung einer ebenen Platte).

10 Geschachtelte Unterprogrammaufrufe im Software-Orientierten Stil

Beim Software-orientierten Programmierstil ist es für einen übersichtlichen Programmentwurf notwendig, dass ein Unterprogramm ein anderes aufrufen kann. Dies bedeutet, dass auch in dem Unterprogramm der nächst tieferen Ebene die Ausführung der Befehle i.d.R. über einen (oder mehrere) Taktzähler verwaltet werden muss. Es muss in allen Unterprogrammebenen derselbe Taktzähler wie im Hauptprogramm verwendet werden, d.h. der Taktzähler muss als Aufrufparameter vom rufenden ans gerufene Unterprogramm übergeben werden. Zusätzlich müssen für die Befehlssteuerung im rufenden Programm die im gerufenen Unterprogramm erreichten Zählerwerte bekannt sein, um im Hauptprogramm eine korrekte Befehlsaktivierung vornehmen zu können. Dies geschieht über die Deklaration des Aufrufparameters als inout.
Für den Fall, dass ein Unterprogramm bedingt, d.h. in Abhängigkeit von den Eingabedaten des Benutzerprogramms aufgerufen wird, müssen das Hauptprogramm und alle darunterstehenden Unterprogramme bei ihren Laufzeitbetrachtungen von der worst case-Situation ausgehen. Diese ist, dass die durchlaufene Aufrufkette im Aufrufbaum maximal lang ist.
Schließlich muss der endliche Automat zum Aufruf jedes Abschnitts (Moduls) stets im Hauptprogramm verbleiben. Der Automat bildet zusammen mit den Deklarationen für den Takt- und Abschnittszähler und den packages für Konstanten, Prozeduren und Bibliotheken das Hauptprogramm, das sonst keine weiteren Befehle enthalten darf.

Zusammengefasst kann gesagt werden, dass solcherart verwaltete Aufrufketten eine hierarchische Programmentwicklung erlauben, und dass die beschriebenen Programmabschnitte unterschiedliche Programmmodule realisieren. Damit können die beiden wichtigsten Elemente strukturierter Programmierung, Hierarchie und Modularität, auch beim Software-orientierten Prorammierstil von VHDL verwendet werden.

11 Besonderheiten des For-Befehls in VHDL

Bei for- und while-Befehlen werden die Unterschiede zwischen VHDL und anderen Programmiersprachen am deutlichsten sichtbar. Ein VHDL-for-Befehl wird für andere Zwecke verwendet als das for in allen anderen prozeduralen Programmiersprachen und dementsprechend auch anders synthetisiert. Diese Erkenntnis ist wichtig.

11.1 Synplify- und xst-Zähler mit for und Zählvariable

Das nachfolgende Programm (Code 16), das in einer prozeduralen Sprache eine Schleife definieren würde, ist zwar auch in VHDL zulässig, stellt aber dort keine Schleife dar, sondern hat einen völlig anderen Effekt.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.My_Package.all;

package My_Package is
  constant HoechsterWertVonj: integer:= 7;
  constant HoechsterWertAmAusgabeport: integer:= HoechsterWertVonj;
  constant HoechsterWertVoni: integer:= 2;
end My_Package;

entity Zaehler is
  Port (clk: in STD_lOGIC;
        NotwendigerAusgabeport: inout integer
         range 0 to HoechsterWertAmAusgabeport);
end Zaehler;

architecture One of Zaehler is
  begin
  process(clk)
    variable j: integer range 0 to HoechsterWertVonj := 1;
     -- j zaehlt 1,3,5,7,1,3,5,7,...
    begin
      if clk'event and clk = '1' then
        for i in 1 to HoechsterWertVoni loop
          j := j+1;
          -- Dieser Befehl ist ein Zaehler, der auch ohne for immer laeuft.
          -- Das for-Statement zaehlt nicht auf HoechsterWertVonj.
          -- Vielmehr wirkt sich for auf das Zaehlinkrement aus: es
          -- wird HoechsterWertVoni statt 1.
         end loop;
        NotwendigerAusgabeport <= j;
        -- Register => wird einen Takt spaeter gueltig. Es erfolgt mit
        -- jedem Takt eine Ausgabe.
```

```
      end if; -- clk'event and clk = '1' ohne else
    end process;
end One;
```

Programmcode 16: Endloszähler trotz endlicher for-Schleife.

Es wird bei obigem Code ein Zähler erzeugt, der unabhängig vom for-Befehl endlos lange zählt. Da der Zähler nur auf einer 3-Bit-Variablen beruht, läuft er ab Zahlen, die größer als 7 sind, periodisch über. Das zeitliche Verhalten nach der Synthese mit Synplify ist in Bild 15 zu sehen. Das Ausgangssignal ist bis auf den Anfangswert identisch mit der Zählvariablen. Wie

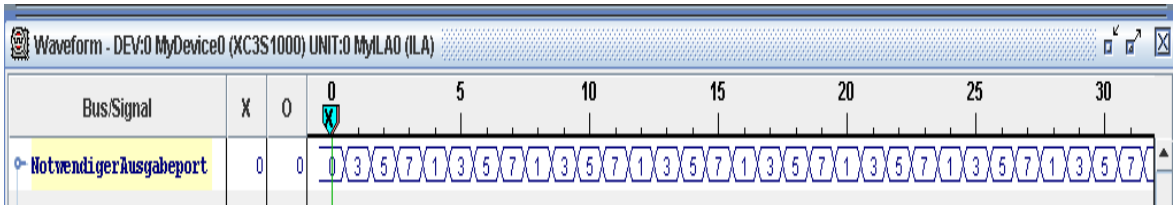

Bild 15: Das zeitliche Verhalten nach der Synthese von Code 16 mit Synplify.

man sieht, wirkt sich die for-Schleife nur auf das Zählerinkrement aus, nicht auf das Zählen selber. Der Grund dafür liegt darin, dass bereits der Compiler das for-Statement auswertet und nicht das FPGA. Es ist ebenfalls der Compiler, der ein erhöhtes Inkrement für den Zähler erzeugt. Der Zählerwert endet deshalb auch nicht mit der Schleifenobergrenze, wie er es bei allen anderen prozeduralen Programmiersprachen tun würde. Das von Synplify erzeugte Kompilat ist in Bild 16 gezeigt. Im wesentlichen ist wieder derselbe Zähler, wie in Code 11 zu

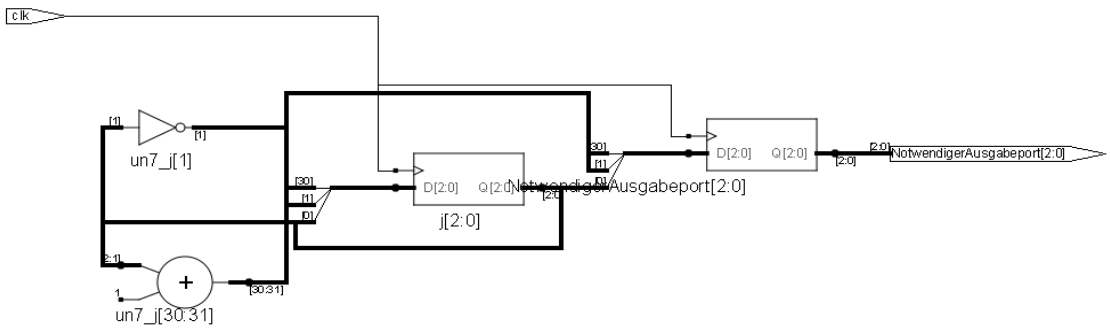

Bild 16: Synthese von Code 16 mit Synplify.

sehen. Neu ist jetzt jedoch, dass jetzt der Addierer zu den 2 Most Significant Bits (MSBs) des Zählers +1 hinzuzählt (ohne Least Significant Bit, LSB), so dass intern mit 0,2,4,6,0,2,4,6,... gezählt wird. Zusätzlich wird das LSB über einen Inverster permanent auf High gesetzt, so dass die in Bild 15 dargestellte ungerade Zählfolge 1,3,5,7,1,3,5,7,... entsteht.
Bei xst wird im starken Gegensatz zu Synplify eine Kette von hintereinandergeschalteten Addierern erzeugt (Bild 17), die jeweils +1 zum Inkrement des Zählers hinzuzählen. D.h., bei xst definiert die Schleifenobergrenze die Anzahl der hintereinander verketteten Addierer. Eine

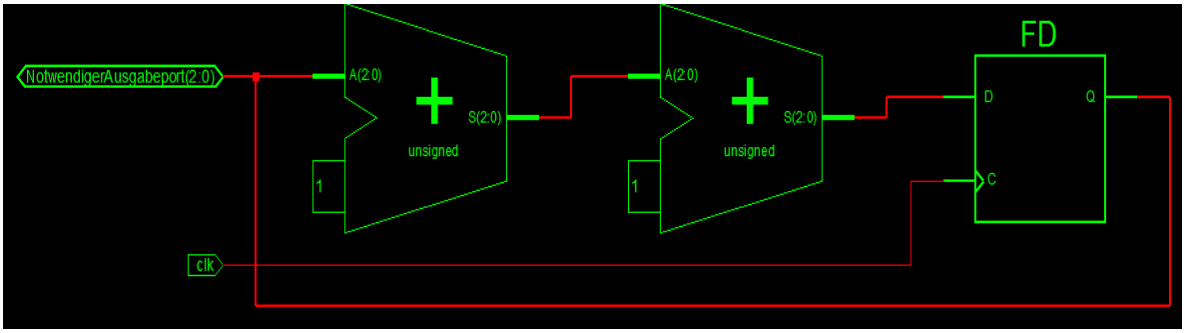

Bild 17: Synthese von Code 16 mit xst.

Schleifenobergrenze von 100 beispielsweise würde in 100 Addierern resultieren, von denen jeder +1 hinzuzählt. Ein eigenes Zählregister gibt es bei der Synthese von Code 16 durch xst nicht, vielmehr wird das Register des Ausgangssignals auch zum Zählen verwendet, was Chip-Ressourcen spart. Das von xst erzeugte Ergebnis ist dennoch bei weitem nicht optimal, da die mehrfache Addition von +1 auch in einem einzigen Schritt erreicht werden könnte. Insgesamt ist das xst-Schaltbild aber übersichtlicher als das von Synplify erzeugte.

11.1.1 Zusammenfassung

Der for-Zähler mit Zählvariable kann modellhaft so interpretiert werden, dass der Schleifenkörper i := i+1; compiler-intern Schleifenobergrenze-Mal hintereinander geschrieben wird, wodurch sich das Inkrement auf den Wert der Schleifenobergrenze erhöht. Aus einem vermeintlichen endlichen for-Zähler wird ein Endloszähler. Der Grund liegt darin, dass for-Statements mit Zählvariablen bereits zur Compile-Zeit ausgewertet und nicht zur Laufzeit ausgeführt werden.

11.2 Verallgemeinerter Synplify- und xst-Zähler mit for und Zählvariable

Das in Code 16 gezeigte Beispiel eines Endloszählers mit for und Zahlvariable kann auf beliebige multiplikative Zählinkremente der Art forSchleifenobergrenze*Inkrement verallgemeinert werden. Dies ist aufwärtskompatibel zum Beispiel in Code 2 und wird exemplarisch in Code 17 gezeigt. Im Beispiel von Code 17 beginnt der Zähler mit einem Anfangswert (=9) und zählt in Einheiten von forSchleifenobergrenze*Inkrement (= 9) solange, bis er überläuft, um dann wieder von vorne zu beginnen. Der Grund dafür, dass wiederum ein Endloszähler gezeigt wird, ist, dass man hierbei die Verallgemeinerung auf beliebige Anfangswerte und auf multiplikative Inkremente besonders leicht fassen kann. Darüberhinaus ist der Unterschied zwischen den beiden Zählvariablen besonders gut darstellbar. Die Zählvariablen in Code 17 heißen jetzt nicht mehr i und j, sondern i und `ForNeutraleZaehlvariable`, um darauf hinzuweisen, dass es sich um zwei völlig verschiedene Variablen handelt, die dennoch von VHDL in impliziter Weise über das Inkrement der zweiten Variablen gekoppelt werden. Diese Kopplung ist eines von vielen Beispielen, die zeigen, dass bei VHDL vieles implizit geschieht.

```
--Package fuer Konstanten und Typdefinition:
package Konstanten is
 constant forSchleifenobergrenze: integer := 3;
```

```
 constant HoechsterWertVonInkrement: integer:= 4;
 constant HoechsterWertVonForNeutraleZaehlvariable: integer:=
  (forSchleifenobergrenze*HoechsterWertVonInkrement)+1; -- +1 als Reserve
 constant HoechsterWertAmAusgabeport: integer:=
  HoechsterWertVonForNeutraleZaehlvariable;
end Konstanten;

--Hauptprogramm
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;
entity Zaehler is
 Port (clk: in STD_lOGIC;
     NotwendigerAusgabeport: inout integer range 0 to HoechsterWertAmAusgabeport);
end Zaehler;
architecture One of Zaehler is
 begin
 process(clk)
  variable ForNeutraleZaehlvariable: integer range 0 to
   HoechsterWertVonForNeutraleZaehlvariable := 9;
  variable Inkrement: integer range 0 to HoechsterWertVonInkrement := 3;
  -- Initialisierung im FPGA
  begin
   if clk'event and clk = '1' then
    for i in 1 to forSchleifenobergrenze loop
     ForNeutraleZaehlvariable := ForNeutraleZaehlvariable + Inkrement;
     -- Dieser Befehl ist ein Zaehler, der auch ohne for immer laeuft.
     -- Das for-Statement zaehlt nicht auf forSchleifenobergrenze.
     end loop;
    NotwendigerAusgabeport <= ForNeutraleZaehlvariable;
    -- Register.
   -- Es erfolgt mit jedem Takt eine Ausgabe.
   end if; -- clk'event and clk = '1' ohne else
  end process;
end One;
```

Programmcode 17: Zähler mit for, Zählvariable und beliebigem Inkrement.

Zu beachten ist ferner, dass der Überlauf von `ForNeutraleZaehlvariable` nicht bereits bei forSchleifenobergrenze*HoechsterWertVonInkrement)+1 (=13) erfolgt, sondern erst bei 15, da der Compiler für die Zählvariable eine 4-Bit-Größe erzeugt. Daraus resultiert dann eine Zählreihenfolge, wie sie in Bild 18 dargestellt ist. Die Zählreihenfolge sieht auf den ersten

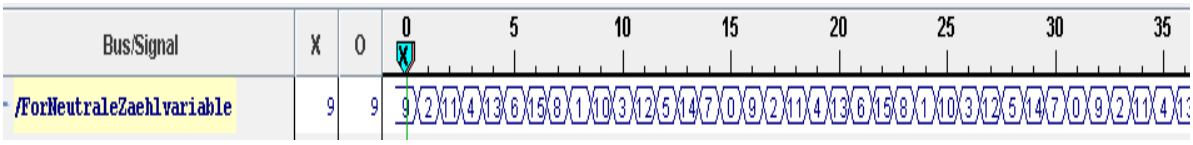

Bild 18: Beispiel für eine Zählreihenfolge bei einem verallgemeinerten Zähler.

Blick zufällig ist, ist aber streng deterministisch.

Xst erzeugt aus Code 17 eine Netzliste, die aufgrund der for-Schleife mit Schleifenobergrenze drei Addierer enthält. Jeder Addierer zählt wiederum die Zahl drei hinzu, so dass insgesamt das in Bild 19 gezeigte Kompilat entsteht.

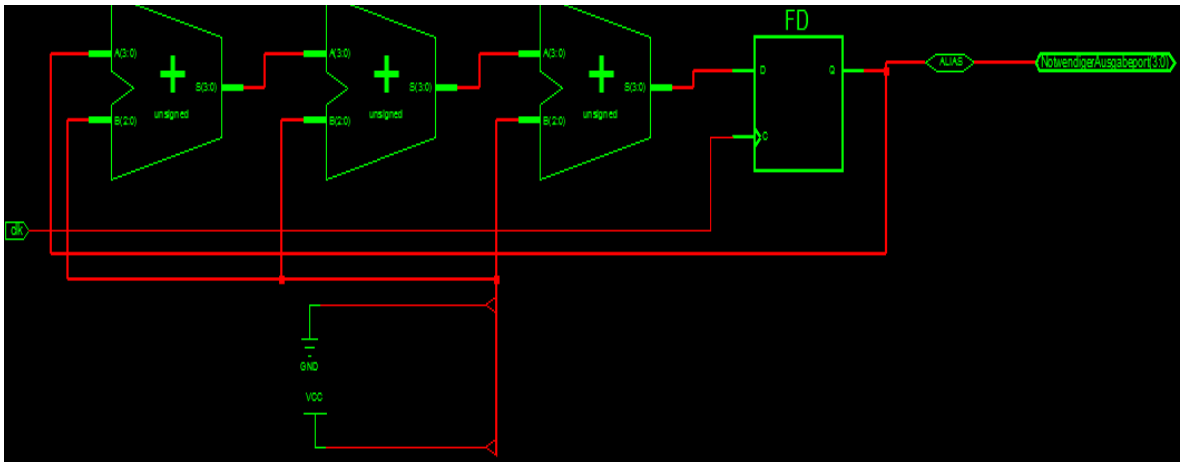

Bild 19: Syntheseergebnis von Code 17 bei xst.

Wichtig ist hier noch festzustellen, dass Modelsim den Code 17 nicht simulieren kann. Dies äußert sich so, dass es nicht mehr terminiert, sondern von Hand beendet werden muss. Leider ist dies kein Einzelfall. Bei vielen weiteren Zählerbeispielen, die im folgenden noch gezeigt werden und sich problemlos synthetisieren lassen, ist Modelsim überfordert. Es sind zwei Dinge, die hier Trost spenden: Zum einen ist das Problem des Nicht-Terminierens leicht erkennbar. Zum anderen ist nach der manuellen Terminierung in den meisten Fällen das dargestellte Zeitdiagramm korrekt.
Viel häufiger tritt allerdings eine Inkompatibilität zwischen Synthese- und Simulatorwerkzeugen dahingehend auf, dass mehr simuliert als synthetisiert werden kann. Das bedeutet: was in der VHDL-Simulation - z.B. über Modelsim von Mentor Graphics - problemlos funktioniert, kann bei der anschließenden Synthese nicht mehr zum gewünschten Resultat führen. Die einzige Möglichkeit, solche Probleme bereits im Vorfeld zu entdecken besteht darin, zu modularisieren und sich den Synthese-Bericht und den erzeugten Schaltplan für jedes Modul genau anzuschauen und zu prüfen, ob wirklich alle Programmbefehle ein Gegenstück in Hardware gefunden haben. Ein umfassender Test nach der Synthese ist ein weiteres probates Mittel, das in der Praxis häufig angewandt wird.

11.2.1 Zusammenfassung

Zusammenfassend kann gesagt werden, dass es nicht nur Inkompatibilitäten zwischen den Synthesewerkzeugen gibt, sondern auch zwischen Synthese- und Simulatorwerkzeugen.

11.3 Synplify- und xst-Zähler mit for und Zählsignal

Ersetzt man die Zählvariable von Code 16 durch ein Zählsignal, während sonst alles gleich bleibt, wird sowohl von Synplify als auch von xst ein ganz anderes Kompilat als in Code 16 erzeugt. Das von xst und von Synplify erzeugte Ergebnis stellt sich dabei für den Anwendungs-

programmierer als überraschend heraus, da die for-Schleife von beiden Werkzeugen komplett ignoriert wird. Synplify erzeugt denselben Endloszähler wie bereits im Schaltplan von Bild 16 angegeben. Der einzige Unterschied ist, dass jetzt ein 3-Bit-Register anstelle eines 2-Bit-Registers generiert wird. Entsprechend erzeugt xst denselben Schaltplan wie in Bild 19, ebenfalls für 3 Bit. Was bei einer Zählvariablen noch zu einer Erhöhung des Inkrements geführt hat, ist bei einem Zählsignal ohne jeden Effekt und führt leider auch zu keiner Fehlermeldung aufgrund der nicht übersetzten for-Schleife! Der einzige Trost ist, dass sich beide Synthesewerkzeuge in gleicher Weise verhalten.

Das bedeutet, dass das Verhalten von for stark davon abhängt, ob als Zähler ein Signal verwendet wird oder nicht. Es wird deshalb empfohlen, for bei Zählern nicht zur multiplikativen Erhöhung des Inkrements zu verwenden. Besser ist es, das benötigte Inkrement explizit anzugeben.

Anhand des Zählers mit for und Zählsignal kann auch auf eine andere Tatsache hingewiesen werden, die wichtig ist, nämlich, dass von Synthesewerkzeugen kommentarlos VHDL-Befehle übergangen werden können. Dieses Phänomen tritt dann auf, wenn vom Compiler zu komplizierte oder nicht vorhergesehene Dinge verlangt werden. Etwas Unvorhergesehenes ist aus der Sicht des Compilers dann gegeben, wenn er einen VHDL-Befehl nicht auf eines seiner Grundelemente wie Zähler, Akkumulator, Multiplexer, Multiplizierer etc. zurückführen kann, oder wenn ein Grundelement wie ein Zähler in einem ihm unbekannten Kontext verwendet wird. Letzteres ist beim Zähler mit for und Zählsignal der Fall. Der Compiler kann nicht ein Zählsignal mit einer for-Schleife kombinieren, um eine automatische Inkrementerhöhung durchzuführen, was bei einer for-Schleife mit Zählvariablen noch problemlos möglich war. Leider wird beim Ignorieren von VHDL-Befehlen von keinem der getesteten Synthesewerkzeug eine Hinweismeldung erzeugt.
Ferner können aus der Sicht des Compilers VHDL-Programme schnell zu kompliziert für die Übersetzung werden. Dafür gibt es im Software-orientierten Programmierstil das Konzept der Module, wodurch zumindest dieses Problem gelöst ist.

11.4 Compiler-basierte Auswertung von for-Statements

VHDL-for-Statements, die nur compiler-interne Variablen benutzen, werden bereits zur Compile-Zeit ausgewertet und nicht zur Laufzeit ausgeführt. Diese Tatsache kann man sich zu Nutze machen und Programme schreiben, die nur noch vom Compiler ausgewertet werden. Code 18 zeigt ein Programm, das kein Taktsignal hat und keine Zeit zur Ausführung benötigt, aber dennoch ein Ergebnis durch den FPGA abliefert.

```
-- Package:
package Konstanten is
  constant forSchleifenobergrenze: integer:= 5;
  constant HoechsterWertVonInkrement: integer:= 4;
  constant HoechsterWertVonForNeutraleZaehlvariable: integer:=
 (forSchleifenobergrenze*HoechsterWertVonInkrement)+1;
 -- +1 als Reserve
  constant HoechsterWertAmAusgabeport: integer:=
   HoechsterWertVonForNeutraleZaehlvariable;
end Konstanten;
```

```
--Hauptprogramm:
use work.Konstanten.all;
entity SchleifeInProzedur is
  port(NotwendigerAusgabeport: out integer range 0 to HoechsterWertAmAusgabeport);
end SchleifeInProzedur;

architecture One of SchleifeInProzedur is

procedure test(variable i: inout integer range 0 to
HoechsterWertVonForNeutraleZaehlvariable;
variable inc: in integer range 0 to HoechsterWertVonInkrement) is
begin
  for j in 1 to forSchleifenobergrenze loop
    i := i + inc;
  end loop;
end procedure;

begin
    process
      variable ForNeutraleZaehlvariable: integer range 0 to
       HoechsterWertVonForNeutraleZaehlvariable;
      variable Inkrement: integer range 0 to HoechsterWertVonInkrement;
      -- eine Initialisierung bei der Deklarierung ist nicht moeglich, weil
      -- diese erst zur Laufzeit auf dem FPGA wirksam wuerde.
      -- Auf dem FPGA wird aber nichts mehr ausgefuehrt.
      begin
        ForNeutraleZaehlvariable := 2;
        -- explizite Initialisierung fuer den Compiler
        -- Schreiben vor dem Lesen, d.h. Compiler-interne Groesse
        Inkrement := 3; -- explizite Initialisierung fuer den Compiler
        -- Schreiben vor dem Lesen, d.h. Compiler-interne Groesse
        test(ForNeutraleZaehlvariable, Inkrement);
        NotwendigerAusgabeport <= ForNeutraleZaehlvariable;
    end process;
end One;
```

Programmcode 18: Auswertung von for-Schleifen zur Compile-Zeit.

Das Ausgabesignal des FPGA wird nach dem Programmladen statisch auf den bereits vom Compiler berechneten Ergebniswert verdrahtet. Im Beispiel von Code 18 wird das in der for-unabhängigen Zählvariablen i angegebene Inkrement (=3) vom Compiler mit der Zahl der Schleifendurchläufe (=5) multipliziert. Außerdem wird der Anfangswert von i (=2) hinzuaddiert. Darüberhinaus wertet der Compiler auch den Prozeduraufruf aus. Der Ergebniswert, den das FPGA ausgibt, beträgt deshalb 17 = 2 +3*5.

Bemerkenswert ist ferner, dass bei Programmen, die ausschließlich vom Compiler ausgewertet werden, die for-Schleife wie in jeder anderen prozeduralen Programmiersprache behandelt wird, d.h. als endliche Schleife. Dies steht im völligen Kontrast zur kombinierten Auswertung durch Compiler und FPGA, wie sie in den vorangegangenen Zählerbeispielen gezeigt wurde und trägt zur Verwirrung des VHDL-Programmierers bei. Erfreulicherweise ist diese Situationen daran erkennbar, dass entweder kein Taktsignal vorhanden ist, oder wenn doch, dass der Compiler eine Warnung ausgibt, dass das Taktsignal nicht verwendet wird. Ein zweiter Hin-

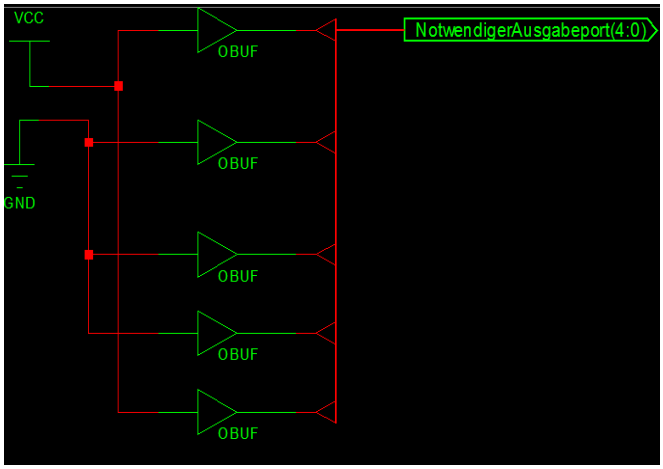

Bild 20: Syntheseergebnis von Code 18.

weis auf eine rein Compiler-basierte Auswertung ist die unendliche schnelle Ausführungszweit (0 Takte Laufzeit). Diese ist bereits bei der Simulation erkennbar.

Zu beachten ist ferner, dass die Compiler-basierte Programmauswertung von Code 18 nicht mehr funktioniert, wenn das Statement i := i + inc; durch den Befehl i <= i + inc; ersetzt wird. Der Grund liegt darin, dass Synthesewerkzeuge für ein Signal stets ein speicherndes Element erzeugen müssen. Speichernde Elemente sind neben Registern auch komplexere Einheiten wie Zähler, Addierer, Akkumulator oder Schieberegister. Diese enthalten nach aussen nicht sichtbar ebenfalls ein Register. Durch die Generierung des speichernden Elements ist dem Compiler jedoch die Möglichkeit genommen, die for-Schleife bereits zur Übersetzungszeit auszuwerten. Darüberhinaus werden die in Code 18 verwendeten Variablen zuerst beschrieben bevor sie gelesen werden, d.h. sie werden auf keinen Fall als Registervariable synthetisiert, sondern bleiben eine Compiler-interne Größe und können dementsprechend bereits zur Übersetzungszeit verarbeitet werden.

Dass mit der Programmausausführung durch den Compiler auch umfangreichere Berechnungen im Sinne eines Software-orientierten Programmierstils gemacht werden können, zeigt das nächste Beispiel (Code 19), in dem die Summe der ersten 9 ganzen Zahlen alleine durch den Compiler berechnet wird. Das so berechnete Ergebnis (=45) wird bei der FPGA-Programmierung in das Chip geladen und dann vom FPGA ausgegeben, bei insgesamt 0 Takten Laufzeit.

```
package Konstanten is
  constant Schleifenobergrenze: integer:= 9;
  constant HoechsterWertVoninc: integer:= 1;
  constant HoechsterWertVoni: integer:=
       Schleifenobergrenze*HoechsterWertVoninc;
  constant HoechsterWertVonSumme: integer:=
       Schleifenobergrenze*Schleifenobergrenze; -- genauer (1/2)*n*(n+1)
  constant HoechsterWertAmAusgabeport: integer:= HoechsterWertVonSumme;
end Konstanten;
use work.Konstanten.all;

entity SummeGanzerZahlen is
  port(NotwendigerAusgabeport: out integer
        range 0 to HoechsterWertAmAusgabeport);
```

```
end SummeGanzerZahlen;

architecture One of SummeGanzerZahlen is
  begin
    process
      variable Akkumulator: integer range 0 to HoechsterWertVoni;
      variable Increment: integer range 0 to HoechsterWertVoninc;
      variable Summe: integer range 0 to HoechsterWertVonSumme;
      -- eine Initialisierung bei der Deklarierung ist nicht möglich, weil
      -- diese erst zur Laufzeit auf dem FPGA wirksam wird.
      -- Auf dem FPGA wird aber nichts mehr ausgeführt.
      begin
        Akkumulator := 0; -- explizite Initialisierung für den Compiler
        -- Schreiben vor dem Lesen, d.h. Akkumulator ist
        -- Compiler-interne Größe
        Increment := 1; -- explizite Initialisierung für den Compiler
        -- Schreiben vor dem Lesen, d.h. Increment ist
        -- Compiler-interne Größe
        Summe := 0; -- explizite Initialisierung für den Compiler
        -- Schreiben vor dem Lesen, d.h. Summe ist
        -- Compiler-interne Größe
        for i in 1 to Schleifenobergrenze loop
          i := i + inc;
          Summe := Summe + i;
        end loop;
        NotwendigerAusgabeport <= Summe;
        -- das Ergebnis ist 45 und wird sofort vom FPGA ausgegeben
    end process;
end One;
```

Programmcode 19: Berechnung der Summe der ersten 9 Zahlen durch den Compiler.

11.4.1 Zusammenfassung

VHDL-for-Statements, die nur compiler-interne Variablen benutzen, werden bereits zur Compile-Zeit ausgewertet und nicht zur Laufzeit ausgeführt. Bei solchen Programmen, die ausschließlich vom Compiler ausgewertet werden, wird die for-Schleife wie in jeder anderen prozeduralen Programmiersprache behandelt, d.h. als endliche Schleife. Dies steht im völligen Kontrast zur kombinierten Auswertung durch Compiler und FPGA.

11.5 Das for-Statement im Hardware-orientierten Programmierstil

Eine weitere Anwendung des for-Statements liegt darin, automatisch mehrere funktionale Einheiten derselben Art zu erzeugen. Diese werden vom Compiler entweder seriell hintereinander oder parallel nebeneinander verkettet. Das nächste Beispiel (Code 20) zeigt die automatische Erzeugung von zwei parallel verketteten Elementen. Es handelt sich um parallgeschaltete Inverter. Die Replizierung funktionaler Einheiten wird beim Hardware-orientierten Programmierstil verwendet.

```
entity ParalleleInverter is
  port(A: in BIT_VECTOR(1 to 2);
       B: out BIT_VECTOR(1 to 2));
end ParalleleInverter;
```

```
architecture One of ParalleleInverter is
  begin
  process (B)
    begin
      for I in A'range loop
        B(I) <= not A(I);
      end loop;
  end process;
```

Programmcode 20: Automatische Erzeugung von zwei parallel geschalteten Invertern.

Die Replizierung funktionaler Einheiten tritt jedoch nicht immer offen zu Tage, da durch den Compiler potentielle Veränderungen im Sinne einer Netzlistenoptimierung vorgenommen werden und z.B. Verkettungen von Schleifenkörpern zu einem einzigen Schleifenkörper mit veränderten Parametern mutieren können.

11.6 Geschachtelte for-Schleifen und for-Schleifen mit variabler Obergrenze

Ineinander geschachtelte for-Schleifen sind beim Hardware-orientierten Programmierstil möglich. Man kann damit automatisch z.B. eine zweidimensionale Anordnung aus parallel und seriell gekoppelten Schleifenkörpern erzeugen. Ein variable Schleifenobergrenze ist bei for in keinem Fall möglich, ebensowenig wie ein wait-statement im Schleifenkörper, da beides zur Laufzeit ausgewertet werden müsste.

11.7 Zusammenfassung

Man kann sich den for-Befehl mit compiler-interner Schleifenvariable modellhaft wie ein Schleifen-Makro in einer üblichen prozeduralen Programmiersprache vorstellen, das den Schleifenkörper so oft repliziert, wie in der Schleifenobergrenze angegeben. Bei einer rein compiler-basierten Auswertung wirkt die for-Schleife mit compiler-interner Schleifenvariable als endliche Schleife. Bei einer gemischten Auswertung durch Compiler und FPGA wirkt die for-Schleife mit compiler-interner Schleifenvariable jedoch so, dass der Compiler das Inkrement eines Zählers implizit um Schleifenobergrenze*Inkrement erhöht (= multiplikative Erhöhung des Inkrements). Bei for-Schleifen mit Zählsignalen wird die Schleife vom Compiler kommentarlos ignoriert. Es wird davon abgeraten, die multiplikative Erhöhung bei Zählern zu verwenden, da diese Funktionalität nur bei Zählvariablen aber nicht bei Zählsignalen funktioniert. Besser ist es, das benötigte Inkrement eines Zählers direkt anzugeben.
Beim Hardware-orientierten Programmierstil kommt zur Schleifenkörper-Replizierung noch eine parallele oder serielle Verkettung der replizierten Schleifenkörper miteinander hinzu.

12 Besonderheiten des While-Befehls in VHDL

12.1 Compiler-basierte Auswertung von while

For und while-Statements sind im Verhalten bis auf eine Ausnahme in all den Fällen identisch, in denen sie durch den Compiler ausgewertet werden. Diese Ausnahme liegt in der Replizie-

rung des Schleifenkörpers in multiple Einheiten, die parallel oder seriell verkettet sind. Dazu ist while nicht in der Lage. Die ansonsten identische Verhaltensweise bei compiler-basierter Auswertung ist allerdings nur dann zu erreichen, wenn berücksichtigt wird, dass jede while-Schleife im Gegensatz zu for einen explizit deklarierten while-Zähler benötigt, der die Zahl der Schleifendurchgänge abzählt.
Der while-Schleifenzähler muss vor Eintritt in die Schleife auf einen Anfangswert gesetzt werden, sonst kann die Schleife nicht terminieren. Zu beachten ist dabei, dass für die Anfangswertinitialisierung die implizite Initialisierung bei der Deklaration mit Hilfe des := Operators nicht funktioniert, denn diese wird erst beim Programmladen in das FPGA wirksam. Das ist für die compiler-basierte Auswertung jedoch zu spät. Statt dessen muss es für den Compiler eine explizite Zuweisung zu Beginn des Programmtexts sowohl für die while-Zählvariable als auch für die while-neutrale Zählvariable geben. Bei der Verwendung von Zählsignalen funktioniert auch dieses Mittel nicht, da von allen Zuweisungen an ein Signal in einem Prozess nur die letzte übrig bleibt.
In Code 21 ist der fachgerechte Umgang mit while für die compiler-basierte Auswertung dargestellt. Das Beispiel ist analog zu Code 18 und hat auch dasselbe Ergebnis (=17) und dieselbe Netzliste wie in Bild 20 bereits gezeigt.

```
-- Package:
package Konstanten is
 constant whileSchleifenobergrenze: integer:= 5;
 constant HoechsterWertVonInkrement: integer:= 4;
 constant HoechsterWertVonwhileNeutraleZaehlvariable: integer:=
  (whileSchleifenobergrenze*HoechsterWertVonInkrement)+1;
 -- +1 als Reserve, um Uberlauf zu vermeiden
 constant HoechsterWertVonwhileZaehlvariable: integer:=
  HoechsterWertVonwhileNeutraleZaehlvariable;
 constant HoechsterWertAmAusgabeport: integer:=
  HoechsterWertVonwhileNeutraleZaehlvariable;
end Konstanten;

--Hauptprogramm:
use work.Konstanten.all;
entity SchleifeInProzedur is
 port(NotwendigerAusgabeport: out integer range 0 to HoechsterWertAmAusgabeport);
end SchleifeInProzedur;

architecture One of SchleifeInProzedur is

procedure test(variable i: inout integer range 0 to
                HoechsterWertVonwhileNeutraleZaehlvariable;
               variable inc: in integer range 0 to HoechsterWertVonInkrement) is
 variable j: integer range 0 to HoechsterWertVonwhileZaehlvariable;
 begin
  j := 1;
  -- explizite Initialisierung fuer den Compiler
  -- Schreiben vor dem Lesen, d.h. Compiler-interne Groesse
  while j <= whileSchleifenobergrenze loop
   i := i + inc;
   j := j + 1;
  end loop;
```

```
end procedure;

begin
  process
   variable whileNeutraleZaehlvariable: integer range 0 to
    HoechsterWertVonwhileNeutraleZaehlvariable;
   variable whileZaehlvariable: integer range 0 to
    HoechsterWertVonwhileZaehlvariable;
   variable Inkrement: integer range 0 to HoechsterWertVonInkrement;
   -- eine Initialisierung bei der Deklarierung ist nicht moeglich, weil
   -- diese erst zur Laufzeit auf dem FPGA wirksam wuerde.
   -- Auf dem FPGA wird aber nichts mehr ausgefuehrt.
   begin
    whileNeutraleZaehlvariable := 2;
    Inkrement := 3;
    -- explizite Initialisierung fuer den Compiler
    -- Schreiben vor dem Lesen, d.h. Compiler-interne Groesse
    test(whileNeutraleZaehlvariable, Inkrement);
    NotwendigerAusgabeport <= whileNeutraleZaehlvariable;
  end process;
end One;
```

Programmcode 21: Analoge Version zu Code 18 mit while statt for.

Das Beispiel von Code 21 berücksichtigt darüberhinaus die Notwendigkeit, die while-Zählvariable oder das while-Zählsignal im Wertebereich so auszulegen, dass am Ende des letzten geplanten Schleifendurchlaufs der Schleifendurchgangszähler nicht überläuft, sonst kann die Schleife ebenfalls nicht terminieren, da der while-Schleifenzähler nach dem Überlauf wieder von vorne beginnt. Bei der Deklaration des Schleifenzählers ist deshalb darauf zu achten, dass eine Reserve im Wertebereich des while-Schleifenzählers vorhanden ist, die mindestens um Eins größer ist als die größte geplante Zahl an Durchläufen.

12.1.1 Zusätzliche Voraussetzungen für eine Compiler-basierte Auswertung

Die weiteren Voraussetzungen für eine Compiler-basierte Auswertung sind die folgenden: es dürfen genau wie bei for keine speichernden Elemente wie Register, Zähler, Akkumulatoren etc. für Variablen erzeugt werden, und es dürfen bis auf Ein- und Ausgangssignale keine weiteren Signale im Code vorkommen, da Signale stets als Register synthetisiert werden. D.h., der Compiler muss in der Lage, sein, den Schleifenkörper komplett mit Hilfe Compiler-interner Variablen zu bearbeiten. Die Bedingung dafür, dass keine Register, Zähler, Akkumulatoren etc. für die Variablen erzeugt werden ist, dass im Programmtext kein Befehl `if clk'event and clk='1'` bzw. kein `wait until clk'event and clk='1'` vorkommt.
Die Compiler-basierte Auswertung von while behandelt dieses Statement wie jede andere prozedurale Programmiersprache auch, im Gegensatz zur gemischten gemischten Auswertung durch Compiler und FPGA. Unter Beachtung dieser Voraussetzungen kann mit while beispielsweise auch die Summe der ersten 9 ganzen Zahlen zur Compilezeit berechnet werden, so wie es mit for bereits gezeigt wurde.

12.1.2 Inkompatibilitäten der Synthesewerkzeuge bei fehlendem Takt

Die Kardinalvoraussetzung zur Compiler-basierten Auswertung ist, dass es keinen Takt gibt bzw. dieser nicht verwendet wird. Die Synthesewerkzeuge verhalten sich bei fehlendem Takt allerdings verschieden. xst weist auf Prozesse mit fehlendem Taktsignal folgendermassen hin: `WARNING:HDLParsers:1406 No sensitivity list and no wait in the process.` Trotz dieser Warnung arbeitet aber der xst-Compiler weiter, denn das Taktsignal ist bei der Compiler-basierten Auswertung nur aus syntaktischen, d.h. formalen Gründen notwendig. Laut VHDL-Sprachdefinition müssen Prozesse entweder eine Sensitivitätsliste oder ein wait-Statement haben. Das Taktsignal wird jedoch bei der Compiler-basierten Auswertung nicht wirklich verwendet. Selbst wenn ein Takt in der port-Deklaration der entity und in der Sensitivitätsliste des Prozesses existiert, benützt ihn der Compiler nicht und zeigt in diesem Fall eine andere Warnung: `WARNING:Xst:647-Input <clk<0>> is never used`.
Man kann bei xst diese Fehlermeldung auch nicht dadurch vermeiden, dass man ein wait-Statement anstelle von `if clk'event and clk='1';` in den Schleifenkörper einfügt. Dann beendet nämlich der xst-Compiler seine Tätigkeit mit der Fehlermeldung `ERROR:Xst:825 - Wait statement in a procedure is not accepted`, obwohl es sich bei der while-Schleife überhaupt nicht um eine Prozedur handelt. D.h., xst erlaubt kein wait im while-Schleifenkörper.

Synplify erfordert im Gegensatz dazu einen Takt oder ein wait-Statement im while-Schleifenkörper, sonst findet keine Compiler-basierte Auswertung statt. Dieses Synthesewerkzeug gibt bei fehlendem Takt folgende Fehlermeldung aus: `@E:CD441:process must contain at least one wait`. Die Fehlermeldung ist allerdings irreführend, denn das Problem lässt sich nicht nur durch Einfügen eines wait-Statements gemäß `wait until clk'event and clk = '1'` am Ende des Schleifenkörpers beheben, sondern -genau wie bei xst auch- durch Einführung eines formalen Taktsignals in der port-Deklaration der entity und in der Sensitivitätsliste des Prozesses. Verwendet man einen formalen Takt und kein wait, erscheint bei Synplify die zu xst analoge Warnung `@W:CD434 : Signal clk in the sensitivity list is not used in the process,` und der Compiler arbeitet erfolgreich weiter. Ein `if clk'event and clk = '1'` im Programmcode ist wie bei xst nicht nötig und würde die Compiler-basierte Auswertung verhindern. Greift man zur Methode des formalen Takts und ignoriert die Fehlermeldung, hat man den Vorteil der Kompatibilität: ein und dasselbe Programm kann von verschiedenen Synthesewerkzeugen übersetzt werden. Bei Einfügen eines wait-Statements am Ende des Schleifenkörpers ist diese Kompatibilität nicht mehr gegeben. Es wird deshalb davon abgeraten, wait bei der Compiler-basierten Auswertung zu verwenden.

12.1.3 Zusammenfassung

Bei der while-Schleifenauswertung nur durch den Compiler verhält sich while genau wie for, bis auf die Replizierung funktionaler Einheiten, die mit while nicht funktioniert. Eine while-Schleife hat beim Compiler soviele Durchläufe, wie in der Schleifenobergrenze angegeben. Es können jedoch bei der Compiler-basierten Auswertung genau wie bei for keine speichernden Elemente wie Register, Zähler, Akkumulatoren etc. für Variablen erzeugt werden, und es dürfen bis auf Ein- und Ausgangssignale keine weiteren Signale im Code vorkommen, da Signale stets als Register synthetisiert werden. D.h., der Compiler muss in der Lage, sein, den Schleifenkörper komplett mit Hilfe Compiler-interner Variablen zu bearbeiten. Die weitere Voraus-

setzung dafür, dass keine Register, Zähler, Akkumulatoren etc. für die Variablen erzeugt werden ist, dass kein Befehl `if clk'event and clk='1'` bzw. `wait until clk'event and clk='1'` vorkommt. Es wird deshalb empfohlen, kein wait-Statement zu verwenden und statt dessen ein formales Taktsignal in der port-Deklaration der entity und in der Sensitivitätsliste des Prozesses vorzusehen. Es muss schließlich berücksichtigt werden, die while-Zählvariable im Wertebereich so auszulegen, dass am Ende aller Schleifendurchläufe der Schleifendurchgangszähler nicht überläuft, sonst kann die Schleife nicht terminieren. Beachtet man alle obigen Regeln, kann die Compiler-basierte Auswertung kompletter VHDL-Programme in Einzelfällen sinnvoll sein. Der Normalfall ist es jedoch nicht, sondern es geht i.d.R. darum, synthesefähigen Code zu erzeugen.

12.2 FPGA-basierte Auswertung von while

Voraussetzung für synthesefähigem Code ist, dass speichernde Elemente wie Register, Zähler, Akkumulatoren etc. erzeugt werden. Dies ist dann der Fall, wenn der Befehl `if clk'event and clk='1'` zu Beginn des Programms, oder wenn der Befehl `wait until clk'event and clk='1'` am Ende des Schleifenkörpers vorkommt.
Die Klärung der Frage, welche Netzliste unter dieser Voraussetzung erzeugt und vom FPGA ausgeführt wird, erfolgt am besten, wie bereits früher bei for geschehen, mit Hilfe von Endlos-zählern bzw. endlichen Zählern. Daran kann man die Funktion von while darstellen und die Unterschiede zwischen for und while aufzeigen. Der Vollständigkeit halber soll an dieser Stelle noch erwähnt werden, dass vom Compiler dann keine Netzliste erzeugt wird, wenn kein while-Zähler in der Schleife existiert (=kein zählendes Element, das die Schleifendurchläufe zählt). Der Compiler gibt in diesem Fall die Meldung aus: *`WARNING:Xst:821-Loop body will iterate zero times.`*

Der Hauptunterscheid zwischen for und while besteht darin, dass es im while-Schleifenkörper von Endloszählern und endlichen Zählern zwei zählende Elemente gibt: einen while-Schleifenzähler, der die Schleifendurchläufe zählt, und einen while-neutralen Zähler, der die eigentliche Zählerfunktion implementiert. Der Letztere ist identisch zum for-neutralen Zähler der früheren Beispiele. Die beiden zählenden Elemente können im Gegensatz zu for sowohl als Variable als auch als Signal ausgeführt werden, wodurch sich insgesamt 4 Varianten ergeben. Bei for hat man nur die Möglichkeit, den for-neutralen Zähler zwischen Variable und Signal zu variieren, denn der for-Schleifenzähler ist implizit als Variable vorgegeben und kann nicht als Signal verwendet werden. Insgesamt muss man bei while-Schleifen die folgenden 4 Fälle unterscheiden:

1.) Zähler mit while-Zählvariable und while-neutraler Zählvariable
2.) Zähler mit while-Zählsignal und while-neutraler Zählvariable
3.) Zähler mit while-Zählvariable und while-neutralem Zählsignal
4.) Zähler mit while-Zählsignal und while-neutralem Zählsignal

Alle vier Fälle werden im folgenden hinsichtlich der erzeugten Netzliste besprochen, und es werden die Gemeinsamkeiten und Unterschiede zu for erklärt. Insbesonders die variierende Anfangswertinitialisierung und die Vermeidung eines Überlaufs beim while-Zähler sind in den nachfolgenden Beispielen zu beachten. Als Vorgriff auf die Ergebnisse der Tests 1)-4) soll an dieser Stelle bereits gesagt werden, dass es im Falle 1) und 3) nicht möglich ist, einen endlichen

Zähler zu erzeugen, und dass dies im Falle 2) und 4) auch nur bei Verwendung von xst gelingt. Die Funktionalität und damit Nützlichkeit von while ist gering. Die beiden endlichen Zähler (Fall 2 und Fall 4) haben ferner eine feste Schleifenobergrenze, die bereits zur Compilezeit bekannt ist.

12.2.1 Fall 1: Endloszähler mit while-Zählvariable und while-neutraler Zählvariable

Code 22 zeigt einen Zähler mit while-Zählvariable und while-neutraler Zählvariable (Fall 1). Der Anfangswert der while-neutralen Zählvariable ist in diesem Beispiel 5, das Inkrement ist 3. Da die Schleifenobergrenze ebenfalls auf 5 festgesetzt ist und die Schleife zur Compile-Zeit ausgewertet wird, wird genau wie bei for die while-neutrale Zählvariable um das Produkt aus Inkrement*Schleifenobergrenze (=15) inkrementiert und es wird endlos gezählt.

```
--Package:
package Konstanten is
 constant whileSchleifenobergrenze: integer:= 5;
 constant HoechsterWertVonInkrement: integer:= 4;
 constant HoechsterWertVonwhileNeutraleZaehlvariable: integer:=
  (whileSchleifenobergrenze*HoechsterWertVonInkrement)+1;
 -- +1 als Reserve, um Uberlauf zu vermeiden
 constant HoechsterWertVonwhileZaehlvariable: integer:=
  HoechsterWertVonwhileNeutraleZaehlvariable;
 constant HoechsterWertAmAusgabeport: integer:=
  HoechsterWertVonwhileNeutraleZaehlvariable;
end Konstanten;

--Hauptprogramm
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;

entity Zaehler is
 Port (clk: in STD_lOGIC;
       NotwendigerAusgabeport: inout integer range 0 to HoechsterWertAmAusgabeport);
end Zaehler;

architecture One of Zaehler is
 begin
 process(clk)
  variable whileZaehlvariable: integer range 0 to
   HoechsterWertVonwhileZaehlvariable;
  -- whileZaehlvariable wird bereits zur Compilezeit ausgewertet
  -- Eine Initialisierung bei der Deklarierung ist nicht moeglich, weil
  -- diese erst zur Laufzeit auf dem FPGA wirksam wuerde.
  variable Inkrement: integer range 0 to HoechsterWertVonInkrement;
  variable whileNeutraleZaehlvariable: integer range 0 to
    HoechsterWertVonwhileNeutraleZaehlvariable :=5;

  begin
   if clk'event and clk = '1' then
    Inkrement := 3;
    whileZaehlvariable := 1;
    -- explizite Initialisierung fuer den Compiler
```

```
    -- Schreiben vor dem Lesen, d.h. i ist Compiler-interne Groesse
    while whileZaehlvariable <= whileSchleifenobergrenze loop
     whileNeutraleZaehlvariable := whileNeutraleZaehlvariable + Inkrement;
     -- Dieser Befehl ist ein Zaehler, der auch ohne while immer laeuft.
     whileZaehlvariable := whileZaehlvariable + 1;
     -- notwendiger while-Zaehler, der zur Compilezeit ausgewertet wird
     NotwendigerAusgabeport <= whileNeutraleZaehlvariable;
     -- Register => wird einen Takt spaeter gueltig. Es erfolgt mit
     -- jedem Takt eine Ausgabe.
    end loop;
   end if; -- clk'event and clk = '1' ohne else
  end process;
end One;
```

Programmcode 22: Endloszähler mit while-Zählvariable und while-neutraler Zählvariable.

Die von xst erzeugte Netzliste enthält Schleifenobergrenze-Mal hintereinandergeschaltete Addierer, die jeweils das Inkrement hinzuzählen, sowie ein Register für den notwendigen Ausgabeport, das gemeinsam mit dem Register für die Zählvariable ist. Bei Synplify wird ebenfalls ein Register für das Ausgangssignal erzeugt. Hinzu kommt ein zweites Register für die Zählvariable und ein Addierer, der in einem Schritt Inkrement*Schleifenobergrenze=15 hinzuaddiert. Wenn die Schleifenobergrenze groß wird, ist der von Synplify erzeugte Code wesentlich Chip-effizienter als der xst-Code.
Die while-neutrale Zählvariable des Beispiels umfasst aufgrund der Konstantendeklaration 5 Bits und läuft bereits bei der 2. Addition über, so dass die in Bild 21 dargestellte Zählreihenfolge entsteht. Insgesamt ist das Verhalten identisch zu dem von for.

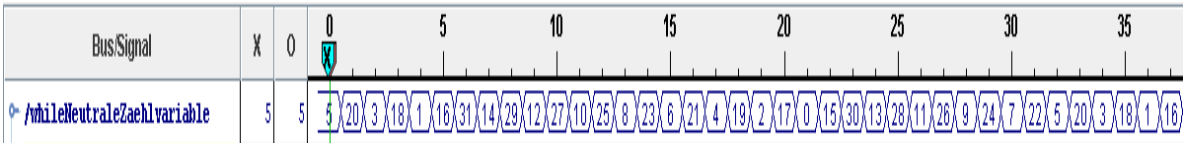

Bild 21: Zählreihenfolge von Code 22 bei xst.

Völlig anders ist die Situation, wenn Synplify zur übersetzung von Code 22 verwendet wird. Dann muss im ansonsten identischen Programm anstelle von `if clk'event and clk = '1' then` am Anfang des Programms der Befehl `wait until clk'event and clk = '1';` am Ende des Schleifenkörpers, d.h. noch innerhalb der Schleife eingesetzt werden, um folgende Fehlermeldung des Compilers zu vermeiden: `@E:CD441:process must contain at least one wait`. Für letzteren Fall erzeugt Synplify einen einzigen Addierer, der das einfache Inkrement von 3 hinzuzählt, sowie ein Register für das Ausgangssignal. Das while-Statement wird also ohne Fehlermeldung komplett ignoriert. Dementsprechend ist auch die Zählweise eine andere, wie Bild 22 zeigt. I
Der Grund dafür ist, dass `wait until..` nicht am Ende des Programms steht und damit analog zu einem `if clk'event...` am Anfang ist, sondern, dass es sich noch innerhalb der Schleife befindet, um die Fehlermeldung des Compilers zu vermeiden. xst erlaubt diese Konstruktion erst gar nicht und bricht mit der etwas irreführenden Fehlermeldung `ERROR:Xst:825 - Wait`

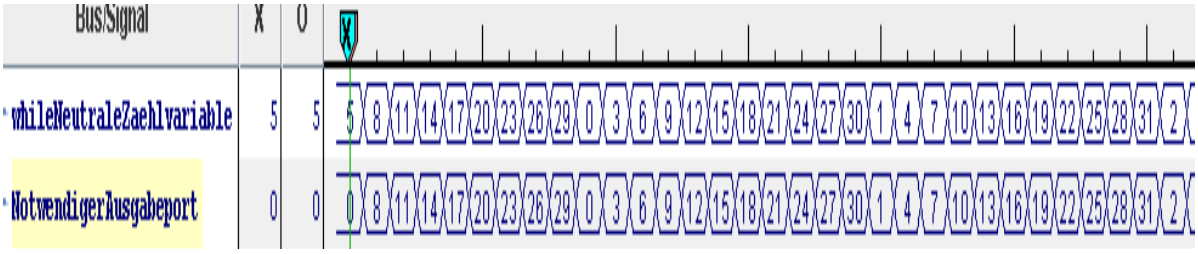

Bild 22: Zählreihenfolge von Code 22 bei Synplify.

`statement in a procedure is not accepted` ab; irreführendend deshalb, weil ein while-Statement keine Prozedur ist.

Genau wie bei der Compiler-basierten Auswertung wird auch bei der FPGA-basierten Auswertung des Codes deshalb davon abgeraten, wait zu verwenden, da ansonsten Inkompatibilitäten bei Synthesewerkzeugen auftreten.

While-Endloszähler sind bzgl. ihrer VHDL-Codierung ähnlich zu for-Endloszählern. Endliche while-Zähler sind jedoch stets von endlichen if-Zählern verschieden, da sie kein `if then else` zum Abprüfen der Schleifenobergrenze haben. Endliche while-Zähler ergeben sich bei einigen Synthesewerkzeugen dann, wenn anstelle einer Zählvariablen ein Zählsignal verwendet wird. Nur in diesem Fall wird für das while-Statement im Gegensatz zum for-Befehl FPGA-Code in Form eines Komparators erzeugt, der zur Laufzeit ausgeführt wird. Der Komparator vergleicht am Ende jedes Schleifendurchlaufs, ob die while-Schleifenobergrenze erreicht ist und führt den Schleifenkörper nur solange aus, wie dies nicht der Fall ist.

12.2.2 Fall 2: Endlicher Zähler mit while-Zählsignal und while-neutraler Zählvariable

Code 23 zeigt einen endlichen Zähler mit while-Zählsignal und while-neutraler Zählvariable. While und for verhalten sich in diesem Falle nicht identisch. Der Grund dafür ist, dass der for-Zählindex nicht als Signal deklariert werden kann, sondern implizit vom Compiler als Variable zur Verfügung gestellt wird, d.h., der hier gezeigte Fall ist mit for nicht möglich.

```
--Package:
package Konstanten is
  constant whileSchleifenobergrenze: integer:= 5;
  constant HoechsterWertVonInkrement: integer:= 6;
  constant HoechsterWertVonwhileNeutraleZaehlvariable: integer:=
   (whileSchleifenobergrenze*HoechsterWertVonInkrement)+1;
  -- +1 als Reserve, um Uberlauf zu vermeiden
  constant HoechsterWertVonwhileZaehlsignal: integer:=
   HoechsterWertVonwhileNeutraleZaehlvariable;
  constant HoechsterWertAmAusgabeport: integer:=
   HoechsterWertVonwhileNeutraleZaehlvariable;
end Konstanten;

--Hauptprogramm
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;

entity Zaehler is
```

```
  Port (clk: in STD_lOGIC;
        NotwendigerAusgabeport: inout integer range 0 to
HoechsterWertAmAusgabeport);
end Zaehler;

architecture One of Zaehler is
  signal whileZaehlsignal: integer range 0 to
   HoechsterWertVonwhileZaehlsignal :=1;
  -- implizite Initialisierung fuer das FPGA
  begin
  process(clk)
    variable Inkrement: integer range 0 to HoechsterWertVonInkrement :=3;
    variable whileNeutraleZaehlvariable: integer range 0 to
     HoechsterWertVonwhileNeutraleZaehlvariable :=8;
    -- implizite Initialisierung fuer das FPGA
    begin
      if clk'event and clk = '1' then
        while whileZaehlsignal <= whileSchleifenobergrenze loop
          whileNeutraleZaehlvariable := whileNeutraleZaehlvariable +
           Inkrement;
          -- ist ein Zaehler, der solange zaehlt, wie die while-Schleife
          -- aktiv ist, d.h. zaehlt
          NotwendigerAusgabeport <= whileNeutraleZaehlvariable;
          -- Register => wird einen Takt spaeter gueltig. Es erfolgt mit
          -- jedem Takt eine Ausgabe.
          whileZaehlsignal <= whileZaehlsignal + 1;
        end loop;
      end if; -- clk'event and clk = '1' ohne else
    end process;
end One;
```

Programmcode 23: Endlicher Zähler mit while-Zählsignal und while-neutraler Zählvariable.

Man kann feststellen: Wenn die while-Zählvariable durch ein Zählsignal ersetzt wird und ansonsten alles wie in Code 22 gleich bleibt, erzeugt xst für das Zählsignal einen echten Zähler, der mit jedem Takt erhöht wird und einen Komparator, der die while-Obergrenze abprüft. Für die while-neutrale Zählvariable wird ein Zähler bestehend aus Addierer und Register generiert, der nur solange zählt, bis die while-Obergrenze erreicht ist. Insgesamt verhält sich hier das while-Statement bei xst so, wie man es von von anderen Programmiersprachen her gewohnt ist.

Synplify hingegen kann den Code 23 nicht übersetzen und bricht mit der Fehlermeldung `@E:CD353:While loop is not terminating?` ab. Ein wait-Statement am Ende des Schleifenkörpers anstelle von `if clk'event...` am Anfang des Programms verhindert zwar den Abbruch, hat aber dieselbe Netzliste und dieselbe Zählweise in Bild 22 zur Folge. D.h., es wird das while-Statement ohne Fehlermeldung komplett ignoriert, und es wird kein endlicher Zähler generiert. Modelsim kann den Code 23 ebenfalls nicht simulieren und muss von Hand abgebrochen werden. Derselbe Befund ergibt sich auch bei Fall 4) „Signal mit Signal“, der noch erläutert wird.

Das bedeutet, dass das Verhalten von while stark vom verwendeten Synthesewerkzeug abhängt, sobald als while-Zähler ein Signal verwendet wird. Es wird deshalb nicht empfohlen, Zähler mit einem while-Zählsignal zu verwenden, um Inkompatibilitäten zu vermeiden.

12.2.3 Fall 3: Endloszähler mit while-Zählvariable und while-neutralem Zählsignal

Wenn die while-neutrale Zählvariable durch ein Zählsignal ersetzt wird, während als while-Index eine Zählvariable verwendet wird, erzeugt xst einen Endloszähler. Weiterhin wird bei xst für das while-neutrale Zählsignal ein Akkumulator erzeugt, der mit jedem Takt nur um das Inkrement erhöht wird, nicht um ein multiplikatives Inkrement. Ist das Inkrement Eins, wird anstelle des Akkumulators ein echter Zähler erzeugt. Hinzu kommt noch ein Register für den notwendigen Ausgabeport, das allerdings unnötig ist, da der Akkumulator den Ausgangswert bereits speichert. Synplify erzeugt in jedem Fall einen Addierer, der ebenfalls nur das Inkrement hinzuzählt. Hinzukommen ein Register für die while-neutrale Zählvariable und eine zweites Register für das Ausgangssignal.
Das while-Statement hat sowohl bei xst als auch bei Synplify keine Auswirkung zur Laufzeit und wird ohne Fehlermeldung ignoriert, d.h., es wirkt sich nicht auf das Inkrement des while-neutralen Zählsignals aus. Dieses Verhalten ist analog zum for-Statement. Die Zählweise ist bei beiden Synthesewerkzeugen gleich und identisch zu der von Modelsim. In diesem Fall verhalten sich also die Synthese- und Simulatorwerkzeuge gleich.

12.2.4 Fall 4: Endlicher Zähler mit while-Zählsignal und while-neutralem Zählsignal

Wenn sowohl die while-Zählvariable als auch die while-neutrale Zählvariable durch ein Zählsignal ersetzt werden, erzeugt xst für beide Signale echte Zähler. Der while-neutrale Zähler bei xst zählt solange, bis die while-Obergrenze erreicht ist. While verhält sich bei xst wie in Fall 2 (while-Zählsignal mit while-neutralem Zählvariable), d.h. wie in anderen Programmiersprachen auch.
While und for sind hier nicht äquivalent. Der Grund dafür ist, dass der for-Zählindex nicht als Signal deklariert werden kann, d.h., der hier gezeigte Fall ist mit for nicht möglich.
Darüberhinaus unterscheiden sich genau wie im Fall 2 die Synthesewerkzeuge erheblich voneinander. Synplify kann das von xst übersetzte Programm nicht übersetzen und bricht mit derselben Fehlermeldung wie im Fall 2 ab. Modelsim kann den Code ebenfalls nicht simulieren und muss von Hand abgebrochen werden.
Das Einfügen von `wait until...` am Ende des Schleifenkörpers hilft bei Synplify auch diesmal. Die Einfügung hat auch dieselben Konsequenzen zur Folge wie bei den Fällen 1,2 und 3, wenn dort `wait until` eingefügt wird. D.h., Synplify verhält sich nach der Einfügung von `wait until...` am Ende des Schleifenkörpers völlig unabhängig den verschiedenen Szenarien der Fälle 1-4. Das bedeutet: sobald bei Synplify ein wait am Ende des Schleifenkörpers eingefügt wird, wird stets dieselbe Netzliste erzeugt und das while-Statement wird kommentarlos ignoriert. Die daraus resultierenden Zähler zählen immer gleich und sind unabhängig von den Fallunterscheidungen 1-4. Insbesonders wird im Gegensatz zu xst kein endlicher Zähler erzeugt.

12.2.5 Ausgabe von Signalwerten mit while

Gegeben sei die folgende Aufgabe: in einer endlichen while-Schleife soll der Wert des while-Schleifenzählers ausgegeben werden. Für den while-Schleifenzähler sind eine Variable oder ein Signal zulässig. Weitere Zähler sind nicht erlaubt.

Zur Lösung dieser Aufgabe wurden alle Varianten getestet. Diese sind: Zählvariable, Zählsignal, xst, Synplify, ohne wait until, mit wait until. In keinem einzigen Fall konnte ein FPGA-Code gefunden werden, der die gestellte Aufgabe löst. Das Einzige, was gut funktioniert hat, ist eine Bildschirmausgabe durch Modelsim über die Bibliotheksroutine fprint bei Verwendung einer while-Zählvariablen. Dafür kann while eingesetzt werden. Am nächsten an die gesuchte Lösung, ohne sie freilich zu erreichen, kam Synplify bei Verwendung einer while-Zählvariablen und von wait until im Schleifenkörper. Synplify konnte in diesem Falle immerhin eine FPGA-Netzliste erstellen, in der die Schleife korrekt hochgezählt und der Wert der while-Zählvariablen richtig auf einem port ausgegeben werden. Das Problem war jedoch, dass nach Erreichen des Stop-Zustandes sich alles periodisch wiederholt hat, d.h. der else-Befehl vor dem null-Statement wurde kommentarlos ignoriert! Modelsim verhielt sich genau wie Synplify, d.h. mangels Terminierung musste man ModelSim manuell beenden. Das dazu gehörende Programm für Synplify ist in Code 24 gezeigt.

```
-- Package:
package Konstanten is
  constant whileSchleifenobergrenze: integer:= 5;
  constant HoechsterWertVonwhileZaehlvariable: integer:=
   whileSchleifenobergrenze+1;
  -- +1 als Reserve, um Überlauf zu vermeiden
  constant HoechsterWertAmAusgabeport: integer:=
   HoechsterWertVonwhileZaehlvariable;
  type Abschnitte is (Abschnitt0, Stop); -- enumeration type
  -- Programm hat nur einen Abschnitt + Stop
end Konstanten;

--Hauptprogramm:
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.Konstanten.all;

entity ifSchleife is
Port (clk: in STD_lOGIC;
      NotwendigerAusgabeport: inout integer range 0 to
       HoechsterWertAmAusgabeport);
end ifSchleife;

architecture One of ifSchleife is
  signal Abschnittszaehler: Abschnitte:= Abschnitt0;
   -- starte mit Abschnitt0
  begin
    process
    variable whileZaehlvariable: integer range 0 to
     HoechsterWertVonwhileZaehlvariable;
    begin
      if Abschnittszaehler = Abschnitt0 then
        whileZaehlvariable := 1;
        while (1 <= whileZaehlvariable) and
          (whileZaehlvariable <= whileSchleifenobergrenze) loop
          NotwendigerAusgabeport <= whileZaehlvariable;
          -- Register => wird einen Takt spaeter gueltig
          whileZaehlvariable := whileZaehlvariable + 1;
          wait until clk'event and clk = '1';
          end loop;
          Abschnittszaehler <= Stop;
```

```
        -- Register. Wird einen Takt spaeter gueltig.
      else -- Abschnittszaehler <= Stop
        -- Programmende erreicht
        null;
      end if; -- Abschnittszaehler = Abschnitt0
  end process;
  end One;
```

Programmcode 24: Näherungsweise Lösung der gestellten Aufgabe durch Synplify.

Die von Synplify erstellte Netzliste ist in Bild 23 dargestellt. Es ist darin kein Mechanismus

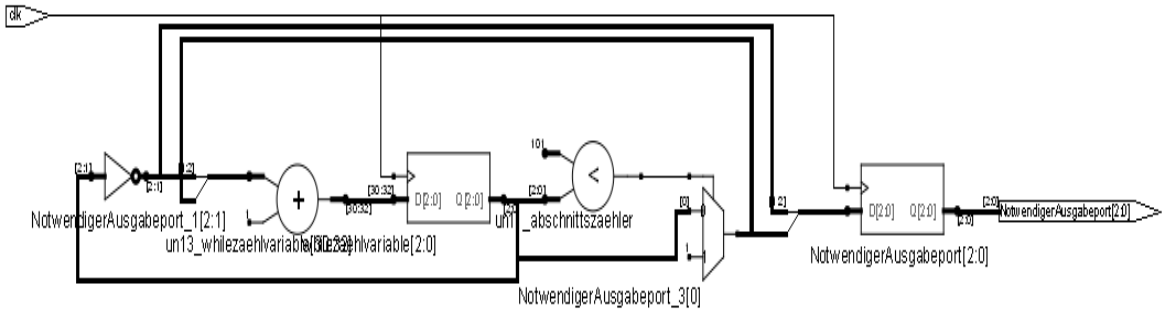

Bild 23: Das von Synplify erzeugte Kompilat von Code 24.

enthalten, der bei Erreichen des Stop-Zustandes, das Register für den Schleifendurchgangszähler oder das Register für die Ausgabe anhält.
Die gestellte Aufgabe ist mit while nicht lösbar, sondern nur mit `if then else`. Wenn man die Aufgabe hingegen modifiziert, indem man 2 Zähler zulässt und zusätzlich mit xst arbeitet, dann ist sie lösbar. Die Lösung entspricht den zuvor beschriebenen Fällen 2 oder 4.

12.2.6 Variable Schleifenobergrenze mit while

Variabel heißt, dass die while-Schleifenobergrenze nicht bereits zur Compilezeit feststeht, sondern erst zur Laufzeit vom FPGA berechnet wird. Synplify und xst wurden daraufhin mit Hilfe von Testprogrammen untersucht. Die Tests wurden so durchgeführt, dass in einem ersten Modul die Summe der ersten n ganzen Zahlen berechnet wurde, wobei n gegeben war. In einem zweiten Modul wurde die vom FPGA berechnete Summe als Obergrenze für einen endlichen Zähler verwendet. Endliche Zähler sind nach dem zuvor Gesagten bei xst für die Fälle 2 und 4 möglich. Es wurden beide Fälle für den zweiten Programmabschnitt getestet. Es konnte jedoch kein Programmbeispiel gefunden werden, bei dem eine variable Obergrenze zu einem korrekten Syntheseergebnis geführt hätte. Eine variable Schleifenobergrenze ist mit while nicht möglich, obwohl ein Komparator erzeugt wird.

12.2.7 Probleme bei for und while

Bei `for ... loop` und `while ... loop` können verschiedene Probleme auftreten: Zum einen kann es vorkommen, dass `for` und `while` vom Synthesewerkzeuge ignoriert werden, ohne dass eine Fehlermeldung ausgegeben wird. Zum anderen gibt es Inkompatibilitäten zwi-

schen den Synthesewerkzeugen untereinander und im Hinblick auf die Simulation. Zum dritten ist die Funktionalität von for und while im Software-orientierten Programmierstil gering und kann vollständig durch einen Zähler zusammen mit `if then else` ersetzt werden. Deshalb werden `for` und `while` beim Software-orientierten Programmierstil nicht verwendet, außer eine rein Compiler-basierte Auswertung ist gewünscht.

12.2.8 Zusammenfassung

Zusammenfassend kann gesagt werden, dass die Voraussetzung für synthesefähigem Code ist, dass speichernde Elemente wie Register, Zähler, Akkumulatoren oder Schieberegister erzeugt werden. D.h., jeder synthesefähiger Code mit while benötigt ein Taktsignal, das nicht nur formal deklariert, sondern auch verwendet wird. Dies ist dann der Fall, wenn der Befehl `if clk'event and clk='1'` zu Beginn des gesamten Programms steht, oder wenn der Befehl `wait until clk'event and clk='1'` am Ende des Schleifenkörpers vorkommt und nicht nur Compiler-interne Größen benutzt werden. Es wird allerdings davon abgeraten, wait am Ende des Schleifenkörper zu verwenden, um Inkompatibilitäten zwischen den Synthesewerkzeugen bei der Programmübersetzung zu vermeiden. Ferner wird davon abgeraten, Zähler mit einem while-Zählsignal zu verwenden, da sie ebenfalls zu Inkompatibilitäten führen. Ohne ein while-Zählsignal kann jedoch kein endlicher Zähler erzeugt werden. Das bedeutet, dass while-Schleifen im Allgemeinfall keine nützliche Funktion haben. Endliche Zähler und damit auch endliche Schleifen sollten besser mit `if then else` als mit while implementiert werden.

13 Beispiele für Software-orientierten Programmentwurf

13.1 Gleichungslöser für eine Wärmeleitungsgleichung Version 2

Das nachfolgende Musterbeispiel für Software-orientierten Programmentwurf ist ein Gleichungslöser für die bereits erläuterte Wärmeleitungsgleichung, der für ein FPGA synthetisierbar ist. D.h., der nachfolgende Code löst in größtmöglicher Geschwindigkeit eine lineare partielle Differentialgleichung in Hardware. Er stellt eine Reimplementierung von Code 15 dar. Im Gegensatz zu diesem Code wird diesmal jedoch eine hierarchische und modulare Programmstruktur verwendet, wie sie auch in anderen Programmiersprachen üblich ist.

```
-- Zeigt, wie man einen Algorithmus, der aus mehreren ineinander geschachtelten
-- Unterprogrammaufrufen besteht, synthetisiert. Ein modularer und hierarchischer
-- Programmentwurf steigert die Übersichtlichkeit und Testbarkeit beträchtlich.
-- Es wird eine einfache Waermeleitungsgleichung einer ebenen Platte mit einer
-- anfaenglichen Waermeeinspeisung am linken Rand berechnet.
-- Die Information, wo nach einem Programmneustart fortzufahren ist, wird in
-- folgenden Zaehler gespeichert: Abschnittszaehler, Zeilenindex, Spaltenindex
-- und Iterationsindex. Diese 4 Zaehler stellen einen verteilten Taktzaehler dar.

--Package fuer Konstanten und Typdefinition:
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package Konstanten is
  type Abschnitte is (Abschnitt0, Ende); -- enumeration type
  -- Programm hat nur einen Abschnitt + Ende
  constant RandTemperatur: integer:=127;
```

```
  -- Legt die Temperatur am linken Rand der Platte zu Beginn der Iteration fest.
  -- Alle Raender haben Null Grad (= keine Waermeeinspeisung)
  constant LaengeDerPlatte: integer:=16;
  -- Platte habe die Laenge 16.
  -- Platte sei rechteckig
  constant BreiteDerPlatte: integer:=8;
  -- Platte habe die Breite 8.
  constant HoechsterWertInLaenge : integer:= LaengeDerPlatte-1;
  -- Zeilenindex zur Adressierung der Platte laeuft von 0 bis LaengeDerPlatte-1
  constant HoechsterWertInBreite : integer:= BreiteDerPlatte-1;
  -- Spaltenindex zur Adressierung der Platte laeuft von 0 bis BreiteDerPlatte-1
  type TypPlatte is array (0 to HoechsterWertInLaenge, 0 to
  HoechsterWertInBreite) of integer range 0 to RandTemperatur;
  -- Platte ist ein 2D-Feld, mit Elementen, die maximal so warm wie die
  -- Einspeisetemperatur werden koennen.
  constant HoechsterWertVonMittelwert: integer:=RandTemperatur;
  -- Legt das hoechste Ergebnis der Mittelwertbildung fest. Platte kann nie heisser
  -- als der Rand werden.
  constant HoechsterWertIterationszaehler: integer:=127;
  -- Iteriere 127 Mal ueber die Waermegleichung
  constant HoechsterWertAmAusgabeport: integer:= RandTemperatur;
  -- Es kann hoechstens der Wert der RandTemperatur ausgegeben werden
end Konstanten;


--Package fuer Prozeduren:
library IEEE;
use IEEE.STD_lOGIC_1164.all;
use IEEE.STD_lOGIC_ARITH.All;
use IEEE.STD_lOGIC_UNSIGNED.All;
use work.Konstanten.all;
use STD.TEXTIO.All;
use work.PCK_FIO.all; -- print package

package Prozeduren is
  file Bildschirm : text open write_mode is "STD_OUTPUT";
  -- Bildschirmausgabe bei Modelsim
procedure debug(
  variable Fall: inout integer range 1 to 9;
  -- gibt an, um welchen Fall es sich handelt
  signal Z: inout integer range 0 to HoechsterWertInLaenge;
  -- gibt die Zeile eines Plattenelements aus
  signal S: inout integer range 0 to HoechsterWertInBreite;
  -- gibt die Spalte eines Plattenelements aus
  variable Mittel: inout integer
    range 0 to HoechsterWertVonMittelwert);
  -- gibt den Wert nach der Mittelung eines Elements aus);

procedure AktualisiereNeuePlatteUndMacheAusgabe(
  signal z: inout integer range 0 to HoechsterWertInLaenge;
  -- gibt die Zeile eines Plattenelements aus
  signal s: inout integer range 0 to HoechsterWertInBreite;
  -- gibt die Spalte eines Plattenelements aus
  variable mittel: inout integer
    range 0 to HoechsterWertVonMittelwert;
  -- gibt den Wert nach der Mittelung eines Elements aus
  variable pneu:inout TypPlatte;
  signal ausg: out integer range 0 to HoechsterWertAmAusgabeport);

procedure GebePlatteAus(
```

```
  signal iter: inout integer range 0 to HoechsterWertIterationszaehler;
  -- Zaehlt die Zahl der Iterationen
  variable p: inout TypPlatte);

procedure Fall1EckeObenLinks(variable Fall: inout integer range 1 to 9;
                             variable P:inout TypPlatte;
                             variable PNeu: inout TypPlatte;
                             signal Z: inout integer range
                               0 to HoechsterWertInLaenge;
                             signal S: inout integer range
                               0 to HoechsterWertInBreite;
                             variable Mittel: inout integer range
                                0 to HoechsterWertVonMittelwert;
                             signal Ausg: out integer range
                                0 to HoechsterWertAmAusgabeport);
procedure Fall2EckeObenRechts(variable Fall: inout integer range 1 to 9;
                              variable P:inout TypPlatte;
                              variable PNeu: inout TypPlatte;
                              signal Z: inout integer range
                                0 to HoechsterWertInLaenge;
                              signal S: inout integer range
                                0 to HoechsterWertInBreite;
                              variable Mittel: inout integer range
                               0 to HoechsterWertVonMittelwert;
                              signal Ausg: out integer range
                                0 to HoechsterWertAmAusgabeport);
procedure Fall3EckeUntenLinks(variable Fall: inout integer range 1 to 9;
                              variable P:inout TypPlatte;
                              variable PNeu: inout TypPlatte;
                              signal Z: inout integer range
                                0 to HoechsterWertInLaenge;
                              signal S: inout integer range
                                0 to HoechsterWertInBreite;
                              variable Mittel: inout integer range
                                0 to HoechsterWertVonMittelwert;
                              signal Ausg: out integer range
                                0 to HoechsterWertAmAusgabeport);
procedure Fall4EckeUntenRechts(variable Fall: inout integer range 1 to 9;
                               variable P:inout TypPlatte;
                               variable PNeu: inout TypPlatte;
                               signal Z: inout integer range
                                 0 to HoechsterWertInLaenge;
                               signal S: inout integer range
                                 0 to HoechsterWertInBreite;
                               variable Mittel: inout integer range
                                 0 to HoechsterWertVonMittelwert;
                               signal Iter: inout integer range
                                 0 to HoechsterWertIterationszaehler;
                               signal Ausg: out integer range
                                 0 to HoechsterWertAmAusgabeport;
                               signal Abschnitt: inout Abschnitte);
procedure Fall5ObererRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                   variable P:inout TypPlatte;
                                   variable PNeu: inout TypPlatte;
                                   signal Z: inout integer range
                                     0 to HoechsterWertInLaenge;
                                   signal S: inout integer range
                                     0 to HoechsterWertInBreite;
                                   variable Mittel: inout integer range
                                     0 to HoechsterWertVonMittelwert;
```

```
                                signal Ausg: out integer range
                                  0 to HoechsterWertAmAusgabeport);
procedure Fall6UntererRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                variable P:inout TypPlatte;
                                variable PNeu: inout TypPlatte;
                                signal Z: inout integer range
                                  0 to HoechsterWertInLaenge;
                                signal S: inout integer range
                                  0 to HoechsterWertInBreite;
                                variable Mittel: inout integer range
                                  0 to HoechsterWertVonMittelwert;
                                signal Ausg: out integer range
                                  0 to HoechsterWertAmAusgabeport);
procedure Fall7LinkerRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                   variable P:inout TypPlatte;
                                   variable PNeu: inout TypPlatte;
                                   signal Z: inout integer range
                                     0 to HoechsterWertInLaenge;
                                   signal S: inout integer range
                                     0 to HoechsterWertInBreite;
                                   variable Mittel: inout integer range
                                     0 to HoechsterWertVonMittelwert;
                                   signal Ausg: out integer range
                                     0 to HoechsterWertAmAusgabeport);
procedure Fall8RechterRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                    variable P:inout TypPlatte;
                                    variable PNeu: inout TypPlatte;
                                    signal Z: inout integer range
                                      0 to HoechsterWertInLaenge;
                                    signal S: inout integer range
                                      0 to HoechsterWertInBreite;
                                    variable Mittel: inout integer range
                                      0 to HoechsterWertVonMittelwert;
                                    signal Ausg: out integer range
                                      0 to HoechsterWertAmAusgabeport);
procedure Fall9InnerePlatte(variable Fall: inout integer range 1 to 9;
                            variable P:inout TypPlatte;
                            variable PNeu: inout TypPlatte;
                            signal Z: inout integer range
                              0 to HoechsterWertInLaenge;
                            signal S: inout integer range
                              0 to HoechsterWertInBreite;
                            variable Mittel: inout integer range
                              0 to HoechsterWertVonMittelwert;
                            signal Ausg: out integer range
                              0 to HoechsterWertAmAusgabeport);
procedure ProzedurWaermegleichung(
  variable Fall: inout integer range 1 to 9;
  signal Abschnitt: inout Abschnitte;
  -- Unterscheidet zwischen Prozeduraufruf und Ende
  signal Zeile: inout integer range 0 to HoechsterWertInLaenge;
  -- Adressiert die Zeile eines Plattenelements
  signal Spalte: inout integer range 0 to HoechsterWertInBreite;
  -- Adressiert die Spalte eines Plattenelements
  variable Platte: inout TypPlatte;
  -- 2D-Feld; uber das iteriert wird. Muss als Register synth. werden.
  variable PlatteNeu: inout TypPlatte;
  -- Wert der Platte nach der Mittelung eines Element.
  -- Muss als Register synth. werden.
  signal Iterationen: inout integer range 0 to HoechsterWertIterationszaehler;
```

```
  -- Zaehlt die Zahl der Iterationen
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport);
  -- notwendiger Ausgabeport);
end Prozeduren;

package body Prozeduren is

procedure debug(
  variable Fall: inout integer range 1 to 9;
  -- gibt an, um welchen Fall es sich handelt
  signal Z: inout integer range 0 to HoechsterWertInLaenge;
  -- gibt die Zeile eines Plattenelements aus
  signal S: inout integer range 0 to HoechsterWertInBreite;
  -- gibt die Spalte eines Plattenelements aus
  variable Mittel: inout integer range 0 to HoechsterWertVonMittelwert)
  -- gibt den Wert nach der Mittelung eines Elements aus)
is
  variable BildschirmZeile: line;
  -- Fuer Modelsim
begin
  fprint(Bildschirm, BildschirmZeile, "Fall=%d\n", fo(Fall));
  fprint(Bildschirm, BildschirmZeile, "Z=%d\n", fo(Z));
  fprint(Bildschirm, BildschirmZeile, "S=%d\n", fo(S));
  fprint(Bildschirm, BildschirmZeile, "Mittel=%d\n", fo(Mittel));
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end debug;

procedure AktualisiereNeuePlatteUndMacheAusgabe(
  signal z: inout integer range 0 to HoechsterWertInLaenge;
  -- gibt die Zeile eines Plattenelements aus
  signal s: inout integer range 0 to HoechsterWertInBreite;
  -- gibt die Spalte eines Plattenelements aus
  variable mittel: inout integer
    range 0 to HoechsterWertVonMittelwert;
  -- gibt den Wert nach der Mittelung eines Elements aus
  variable pneu:inout TypPlatte;
  signal ausg: out integer range 0 to HoechsterWertAmAusgabeport) is
begin
  -- Aktualisiere NeuePlatte an der Stelle [z, s]
  pneu(z,s) := mittel;
  -- Register; da Zuweisung nur im if-Zweig; aber nicht im else-Zweig.
  -- Wird einen Takt spaeter gueltig.
  -- Gebe die Platte an der Stelle [z, s]) aus.
  ausg <= mittel;
  -- Register. Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte. Die Werte von pneu und ausg werden erst nach
  -- Terminierung der Prozedur gueltig.
end AktualisiereNeuePlatteUndMacheAusgabe;

procedure GebePlatteAus(-- Fuer Modelsim
  signal iter: inout integer range 0 to HoechsterWertIterationszaehler;
  -- Zaehlt die Zahl der Iterationen
  variable p: inout TypPlatte) is
  -- 2D-Feld; uber das iteriert wird. Muss als Register synth. werden.
  -- Gebe bei Modelsim die Platte am Anfang und nach jeder 8. Iteration,
  -- sowie am Ende der Erhoehungen des Iterationszaehlers
  -- auf dem Bildschirm aus.
  variable BildschirmZeile: line;
```

```
  begin
  if Iter=0 or (Iter mod 7) = 0 or
    Iter=HoechsterWertIterationszaehler-1
    -- Der Wert von Iter ist an dieser Stelle immer um eins kleiner, als der
    -- aktuelle Stand von p, da die Zuweisung an Iter im rufenden Programm noch
    -- nicht wirksam wurde. d.h. die Aktualitaet von Iter und p sind nicht gleich.
    then
    fprint(Bildschirm, BildschirmZeile, "iter=%d\n", fo(iter));
    fprint(Bildschirm, BildschirmZeile, "p:\n");
    for i in 0 to HoechsterWertInLaenge loop -- Gehe uber alle Zeilen
      for j in 0 to HoechsterWertInBreite loop -- Gehe uber alle Spalten
        fprint(Bildschirm, BildschirmZeile," %d", fo(p(i,j)));
      end loop; -- j
      fprint(Bildschirm, BildschirmZeile, "\n");
    end loop; -- i
  else -- (iter mod 7) /= 0
    null; -- gebe bei allen anderen Iterationen nichts aus
  end if; -- (iter mod 7) = 0
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end GebePlatteAus;

procedure Fall1EckeObenLinks(variable Fall: inout integer range 1 to 9;
                             variable P:inout TypPlatte;
                             variable PNeu: inout TypPlatte;
                             signal Z: inout integer range
                               0 to HoechsterWertInLaenge;
                             signal S: inout integer range
                               0 to HoechsterWertInBreite;
                             variable Mittel: inout integer range
                               0 to HoechsterWertVonMittelwert;
                             signal Ausg: out integer range
                               0 to HoechsterWertAmAusgabeport) is
begin
  Mittel := (P(Z+1,S)+ --   Sued
             P(Z,S+1))/2; --Ost
  Fall := 1;
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  S <= S+1;
  -- Synth. Spaltenzaehler
  -- Aktualisiere Spaltenzaehler. Gehe durch die erste Zeile
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall1EckeObenLinks;

procedure Fall2EckeObenRechts(variable Fall: inout integer range 1 to 9;
                              variable P:inout TypPlatte;
                              variable PNeu: inout TypPlatte;
                              signal Z: inout integer range
                                0 to HoechsterWertInLaenge;
                              signal S: inout integer range
                                0 to HoechsterWertInBreite;
                              variable Mittel: inout integer range
                                0 to HoechsterWertVonMittelwert;
                              signal Ausg: out integer range
                                0 to HoechsterWertAmAusgabeport) is
begin
  Mittel := (P(Z,S-1)+ -- West
```

```
            P(Z+1,S))/2; --   Sued
  -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in   Sued und West.
  Fall := 2;
  debug(Fall, Z, S, Mittel);
  -- Aktualisiere jetzt die Platte an der Stelle [Z; S]
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  -- Bringe am Ende einer Zeile den Zeilenzaehler auf den neuesten Stand
  Z <= Z+1; -- gehe zur nachsten Zeile
  -- Synth. Zeilenzaehler
  -- Wird einen Takt spaeter gueltig.
  S <=0; -- beginne wieder mit der ersten Spalte
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall2EckeObenRechts;

procedure Fall3EckeUntenLinks(variable Fall: inout integer range 1 to 9;
                              variable P:inout TypPlatte;
                              variable PNeu: inout TypPlatte;
                              signal Z: inout integer range
                                0 to HoechsterWertInLaenge;
                              signal S: inout integer range
                                0 to HoechsterWertInBreite;
                              variable Mittel: inout integer range
                                0 to HoechsterWertVonMittelwert;
                              signal Ausg: out integer range
                               0 to HoechsterWertAmAusgabeport) is
begin
  Mittel := (P(Z,S+1)+ -- Ost
            P(Z-1,S))/2; -- Nord
  -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in Nord und Ost.
  Fall := 3;
  debug(Fall, Z, S, Mittel);
  -- Aktualisiere jetzt die Platte an der Stelle [Z; S]
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  S <= S+1; -- gehe durch die letzte Zeile
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall3EckeUntenLinks;

procedure Fall4EckeUntenRechts(variable Fall: inout integer range 1 to 9;
                               variable P:inout TypPlatte;
                               variable PNeu: inout TypPlatte;
                               signal Z: inout integer range
                                 0 to HoechsterWertInLaenge;
                               signal S: inout integer range
                                 0 to HoechsterWertInBreite;
                               variable Mittel: inout integer range
                                 0 to HoechsterWertVonMittelwert;
                               signal Iter: inout integer range
                                 0 to HoechsterWertIterationszaehler;
                               signal Ausg: out integer range
                                 0 to HoechsterWertAmAusgabeport;
                               signal Abschnitt: inout Abschnitte) is
begin
  Mittel := (P(Z,S-1)+ -- West
            P(Z-1,S))/2; -- Nord
  -- Neuer Wert ist Mittelwert seiner 2 Nachbarn in Nord und West.
  Fall := 4;
```

```
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  -- Letzte Zeile und letzte Spalte sind erreicht
  -- Eine Iteration ist durchgelaufen
  -- Mache Aufraeumarbeiten am Ende jeder Iteration
  -- Dieser Fall tritt nur einmal alle LaengeDerPlatte*BreiteDerPlatte
  -- Aufrufe auf.
  -- Beginne wieder mit der ersten Zeile und der ersten Spalte fuer die
  -- naechste Iteration.
  Z <=0;
  S <=0;
  -- Wird einen Takt spaeter gueltig.
  -- Kopiere nach jeder Iteration PlatteNeu nach Platte; um Platte auf den
  -- neuesten Stand zu bringen.
  P := PNeu; -- Kopiert ganzes Feld in einem Befehl
  -- Wert wird einen Takt spaeter gueltig.
  -- Bringe am Ende jeder Iteration den Iterationszaehler auf den
  -- neuesten Stand
  if Iter<HoechsterWertIterationszaehler then
    Iter <= Iter+1;-- synth. IterationsZaehler.
    -- Wert wird einen Takt spaeter gueltig.
  else -- Iter=HoechsterWertIterationszaehler
    -- Iterationen hat den HoechsterWertIterationszaehler erreicht. Erhoehe
    -- Iterationen nicht weiter.
    -- Beende Programm
    Abschnitt <= Ende;
    -- Wird einen Takt spaeter gueltig.
    -- Programmende erreicht
  end if; -- Iter<HoechsterWertIterationszaehler
  GebePlatteAus(Iter, P); -- nur bei Modelsim
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall4EckeUntenRechts;

procedure Fall5ObererRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                   variable P:inout TypPlatte;
                                   variable PNeu: inout TypPlatte;
                                   signal Z: inout integer range
                                     0 to HoechsterWertInLaenge;
                                   signal S: inout integer range
                                     0 to HoechsterWertInBreite;
                                   variable Mittel: inout integer range
                                     0 to HoechsterWertVonMittelwert;
                                   signal Ausg: out integer range
                                     0 to HoechsterWertAmAusgabeport) is
begin
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in Sued; Ost
  -- und West. Da nicht durch 3 dividiert werden kann; wird nur West und Ost
  -- verwendet und naherungsweise durch 2 dividiert
  Mittel := (P(Z,S-1)+ -- West
             P(Z,S+1))/2; -- Ost
  Fall := 5;
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  S<=S+1; -- bleibe in der ersten Zeile
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall5ObererRandOhneEcken;
```

```
procedure Fall6UntererRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                    variable P:inout TypPlatte;
                                    variable PNeu: inout TypPlatte;
                                    signal Z: inout integer range
                                      0 to HoechsterWertInLaenge;
                                    signal S: inout integer range
                                      0 to HoechsterWertInBreite;
                                    variable Mittel: inout integer range
                                      0 to HoechsterWertVonMittelwert;
                                    signal Ausg: out integer range
                                      0 to HoechsterWertAmAusgabeport) is
begin
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in Nord; Ost
  -- und West. Da nicht durch 3 dividiert werden kann; wird nur West und Ost
  -- verwendet und naherungsweise durch 2 dividiert
  Mittel := (P(Z,S-1)+ -- West
             P(Z,S+1))/2; -- Ost
  Fall := 6;
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  S<=S+1; -- bleibe in der letzten Zeile
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall6UntererRandOhneEcken;

procedure Fall7LinkerRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                   variable P:inout TypPlatte;
                                   variable PNeu: inout TypPlatte;
                                   signal Z: inout integer range
                                     0 to HoechsterWertInLaenge;
                                   signal S: inout integer range
                                     0 to HoechsterWertInBreite;
                                   variable Mittel: inout integer range
                                     0 to HoechsterWertVonMittelwert;
                                   signal Ausg: out integer range
                                     0 to HoechsterWertAmAusgabeport) is
begin
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in Nord; Sued
  -- und Ost. Da nicht durch 3 dividiert werden kann; wird nur Nord und Sued
  -- verwendet und naherungsweise durch 2 dividiert
  Mittel := (P(Z-1,S)+ -- Nord
             P(Z+1,S))/2; -- Sued
  Fall := 7;
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  S<=S+1; -- bleibe in der Zeile
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall7LinkerRandOhneEcken;

procedure Fall8RechterRandOhneEcken(variable Fall: inout integer range 1 to 9;
                                    variable P:inout TypPlatte;
                                    variable PNeu: inout TypPlatte;
                                    signal Z: inout integer range
                                      0 to HoechsterWertInLaenge;
                                    signal S: inout integer range
                                      0 to HoechsterWertInBreite;
                                    variable Mittel: inout integer range
```

```
                                    0 to HoechsterWertVonMittelwert;
                                  signal Ausg: out integer range
                                    0 to HoechsterWertAmAusgabeport) is
begin
  -- Neuer Wert ist eigentlich der Mittelwert seiner 3 Nachbarn in Nord; Sued
  -- und West. Da nicht durch 3 dividiert werden kann; wird nur Nord und Sued
  -- verwendet und naherungsweise durch 2 dividiert
  Mittel := (P(Z-1,S)+ -- Nord
             P(Z+1,S))/2; -- Sued
  Fall := 8;
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  -- Alle Spalten einer Zeile sind durchgelaufen
  Z<=Z+1; -- gehe in die nachste Zeile
  -- Wird einen Takt spaeter gueltig.
  S <= 0; -- beginne wieder mit der ersten Spalte
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall8RechterRandOhneEcken;

procedure Fall9InnerePlatte(variable Fall: inout integer range 1 to 9;
                            variable P:inout TypPlatte;
                            variable PNeu: inout TypPlatte;
                            signal Z: inout integer range
                              0 to HoechsterWertInLaenge;
                            signal S: inout integer range
                              0 to HoechsterWertInBreite;
                            variable Mittel: inout integer range
                              0 to HoechsterWertVonMittelwert;
                            signal Ausg: out integer range
                              0 to HoechsterWertAmAusgabeport) is
begin
  Mittel := (P(Z-1,S)+P(Z+1,S) + P(Z,S-1)+P(Z,S+1))/4;
  -- Neuer Wert ist Mittelwert seiner 4 Nachbarn Nord; Sued; West und Ost.
  Fall := 9;
  debug(Fall, Z, S, Mittel);
  AktualisiereNeuePlatteUndMacheAusgabe(Z, S, Mittel, PNeu, Ausg);
  S<=S+1; -- bleibe in der Zeile
  -- Wird einen Takt spaeter gueltig.
  -- In dieser Prozedur wird kein Takt ausgewaehlt. Die Ausfuehrungszeit ist
  -- deshalb Null Takte.
end Fall9InnerePlatte;

procedure ProzedurWaermegleichung(
  variable Fall: inout integer range 1 to 9;
  signal Abschnitt: inout Abschnitte;
  signal Zeile: inout integer range 0 to HoechsterWertInLaenge;
  signal Spalte: inout integer range 0 to HoechsterWertInBreite;
  variable Platte:inout TypPlatte ;
  variable PlatteNeu: inout TypPlatte;
  signal Iterationen: inout integer range 0 to HoechsterWertIterationszaehler;
  signal Ausgabe: out integer range 0 to HoechsterWertAmAusgabeport) is
  variable Mittelwert: integer range 0 to HoechsterWertVonMittelwert:=0;
  -- Enthaelt die Mittelung ueber die Nachbarpunkte in Nord; Sued; West und Ost
  begin
    -- Mache alle gewunschten Interationen
    -- Strukturiere jede Iteration
    if Iterationen<= HoechsterWertIterationszaehler then
      -- Maximale Zahl der Iterationen ist noch nicht ueberschritten
```

```
      -- Mache eine Iteration
      -- Behandle sowohl die innere Platte als auch die Rander
      -- Behandle die 4 Eckpunkte und die 4 Rander extra
      -- Behandle alle 9 Falle von Mittelwertbildungen uber die Nachbarpunkte
      -- Der Zeilenindex soll langsamer als der Spaltenindex laufen
      if Zeile=0 and Spalte=0 then
        Fall1EckeObenLinks(Fall, Platte, PlatteNeu, Zeile, Spalte, Mittelwert,
         Ausgabe);
      elsif Zeile=0 and Spalte=HoechsterWertInBreite then
        -- Fall 2: Ecke oben rechts
        Fall2EckeObenRechts(Fall, Platte, PlatteNeu, Zeile, Spalte, Mittelwert,
         Ausgabe);
      elsif Zeile=HoechsterWertInLaenge and Spalte=0 then
        -- Fall 3: Ecke unten links
        Fall3EckeUntenLinks(Fall, Platte, PlatteNeu, Zeile, Spalte, Mittelwert,
         Ausgabe);
      elsif Zeile=HoechsterWertInLaenge and Spalte=HoechsterWertInBreite then
        -- Fall 4: Ecke unten rechts. Eine Iteration zu Ende.
        Fall4EckeUntenRechts(Fall, Platte, PlatteNeu, Zeile, Spalte, Mittelwert,
          Iterationen, Ausgabe, Abschnitt);
      elsif Zeile=0 and 0<Spalte and Spalte<HoechsterWertInBreite then
        -- Fall 5: oberer Rand ohne Ecken
        Fall5ObererRandOhneEcken(Fall, Platte, PlatteNeu, Zeile, Spalte,
Mittelwert,
          Ausgabe);
      elsif Zeile=HoechsterWertInLaenge and 0<Spalte and
        Spalte<HoechsterWertInBreite then
        -- Fall 6: unterer Rand ohne Ecken
        Fall6UntererRandOhneEcken(Fall, Platte, PlatteNeu, Zeile, Spalte,
Mittelwert,
          Ausgabe);
      elsif 0<Zeile and Zeile<HoechsterWertInLaenge and Spalte=0 then
        -- Fall 7: linker Rand ohne Ecken
        Fall7LinkerRandOhneEcken(Fall, Platte, PlatteNeu, Zeile, Spalte,
Mittelwert,
          Ausgabe);
      elsif 0<Zeile and Zeile<HoechsterWertInLaenge and
        Spalte=HoechsterWertInBreite then
        -- Fall 8: rechter Rand ohne Ecken
        Fall8RechterRandOhneEcken(Fall, Platte, PlatteNeu, Zeile, Spalte,
Mittelwert,
          Ausgabe);
      elsif 0<Zeile and Zeile<HoechsterWertInLaenge and
        0<Spalte and Spalte<HoechsterWertInBreite then
        -- Fall 9: im Inneren der Platte
        Fall9InnerePlatte(Fall, Platte, PlatteNeu, Zeile, Spalte, Mittelwert,
         Ausgabe);
      end if; -- Zeile=0 and Spalte=0. Fall 1: Ecke oben links.
      -- alle 9 Kombinationen von Zeile und Spalte wurden berucksichtigt
      else -- Iterationen>=HoechsterWertIterationszaehler
        -- Letzte Iteration wurde durchgefuehrt. Programm wurde in Fall 4 beeendet.
       null; -- mache hier nichts mehr
    end if; -- Iterationen <= HoechsterWertIterationszaehler
    -- In dieser Prozedur werden Takt ausgewaehlt. Die Ausfuehrungszeit ist
    -- deshalb >Null Takte.
end ProzedurWaermegleichung;
end Prozeduren;


-- Hauptprogramm
```

```
library IEEE;
use IEEE.STD_lOGIC_1164.All;
use work.Konstanten.all;
use work.Prozeduren.all;

entity Algorithmus3 is
  Port (clk: in STD_lOGIC;
    NotwendigerAusgabeport: inout integer range 0 to
    HoechsterWertAmAusgabeport);
end Algorithmus3;

architecture One of Algorithmus3 is
  --Deklarationen fuer Signale
  signal Abschnittszaehler: Abschnitte:= Abschnitt0;
  -- starte mit Abschnitt0
  signal Zeilenindex: integer range 0 to HoechsterWertInLaenge:=0;
  -- Zeilenindex fuer die Adressierung der Platte. Beginne mit Zeile 0. Der Rand
  -- der Platte wird extra behandelt.
  signal Spaltenindex: integer range 0 to HoechsterWertInBreite:=0;
  -- Spaltenindex fuer die Adressierung der Platte. Beginne mit Spalte 0. Der Rand
  -- der Platte wird extra behandelt.
  -- Der Spaltenindex soll sich schneller aendern als der Zeilenindex
  signal Iterationsindex: integer range 0 to HoechsterWertIterationszaehler :=0;
  -- starte mit Iteration 0
begin

  process(clk)
    variable Platte: TypPlatte:=(others=>(RandTemperatur,others=>0));
    -- Am linken Rand ist die Temperatur zu Beginn = RandTemperatur sonst 0.
    -- RandTemperatur hat Spaltenindex 0. RandTemperatur klingt ab.
    variable NeuePlatte: TypPlatte:=(others=>(others=>0));
    -- NeuePlatte=Platte nach einer Iteration. Sie hat vor der Iteration die
    -- Anfangswerte 0.
       variable Fall: integer range 1 to 9;
    begin
    if clk'event and clk = '1' then
      if Abschnittszaehler = Abschnitt0 then
       ProzedurWaermegleichung(Fall, Abschnittszaehler, Zeilenindex, Spaltenindex,
          Platte, NeuePlatte, Iterationsindex,
          NotwendigerAusgabeport);
      else -- Abschnittszaehler = Ende
        -- Programmende erreicht
        -- Hier kann nichts mehr gemacht werden
        null;
    end if; -- Abschnittszaehler = Abschnitt0
    end if; -- clk'event and clk = '1'
    -- In diesem Prozess werden Takt ausgewaehlt. Die Ausfuehrungszeit ist
    -- deshalb >Null Takte.
  end process;

end One;
```

Programmcode 25: Wärmeleitung einer ebenen Platte im Software-orientierten Programmierstil.

13.2 CarRing II-Rechnernetzprotokoll

Der nachfolgende Code 26 stellt einen Teil der ISO-Schicht 1c von CarRing II dar. Hieran kann man erkennen, dass die Implementierung von Rechnernetztprotokollen mit Hilfe des Software-orientierten Programmierstils in VHDL möglich ist. Das Vorgehen ähnelt stark der Art und Weise, wie man diese Schicht in einer prozeduralen Programmiersprache implementieren würde. Das zu Beginn gesteckte Ziel im Projekt „Software-orientierte Programmierstil“, auf einfache Weise Rechnernetzprotokolle in ein FPGA synthetisieren zu können, wurde somit erreicht.

```
-- Dateiname: Einmalige Datenuebergabe S u E V14.fm
-- Autor: HRI 16.02.11
-- Titel: Sender L1c von CarRing II
-- Beschreibung:
-- Dieses Programm bezieht sich auf das Dokument:
-- H. Richter, Beschreibung von Local Link, HR 08.11.10
-- Es wird der Fall "Senden weniger Daten (einmalige Datenuebergabe)"
implementiert.
-- In einem Testprogramm werden 3 Worte der Wortbreite n=2 Byte an
-- eine Sendeprozedur uebergeben
-- Es wird der Inter Frame Gap (IFG) berucksichtigt, in dem das LL IF
-- nicht einsatzbereit ist.
-- Der endliche Automat hat 4 Phasen => Das Senden eines Wortes dauert 4 Takte


--Package fur Konstanten:

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package Konstanten is
  type SenderAbschnitte_Typ is (LLIFInitialisieren, DrucknopfAbwarten,
EinWortSenden,
   StatusPruefen, Stop); -- enumeration type
  -- Hauptprogramm hat 4 Abschnitt + Stop
  -- Konstanten fuer das LL IF
  type SendePhase_Typ is
   (PruefenLLIF,Einmalige_Datenuebergabe, SendePause1,
HauptprogrammWeiterschalten);
  -- SendePhase ist ein EA mit Ein- und Ausgabe, der das Timing-Diagramm
  -- gemaess Bild 6 von obigem Dokument umsetzt.
  -- Eingabe in den EA ist USER_CLK und H_TX_DST_RDY_N. Ausgaben sind die
  -- Signale der Eingabeseite des LL IF, d.h.
  -- H_TX_D, H_TX_REM, H_TX_SOF_N, H_TX_EOF_N, H_TX_SRC_RDY_N
  -- Nach der Initialisierung von Aurora wird der EA wird in 4 Takten
  -- durchlaufen, inkl. von 2 Takten Sendepause.
  -- Pro 4 Takte wird bei dieser Implementierung ein 16-Bit Wort gesendet.
  constant kWorteZumSenden: integer:= 3; -- Sende k=3 Worte
  constant WortbreiteBus1: integer:= 2;
  --Bus 1 hat die Wortbreite n=2 gemaess S. 24 von UG224.pdf
  constant BitbreiteBus1: integer:= WortbreiteBus1*8;
  --Bus 1 hat 16 Bit Wortbreite
  --Bus 2 hat 1 Bit Wortbreite und kann den Wert 0 oder 1 an das LL IF
  -- liefern
```

```
  constant NofRestlicheBytesMinusEins: std_logic:='1';
  -- Der Rest, der auf Bus 2 angegeben wird, ist NofRestlicheBytesMinusEins,
  -- d.h. das maximal Mogliche
  -- Allgemeine Charakterisierung des Sendeworts
  -- Das, was gesendet werden soll, ist ein 16-Bit Wort
  -- Die 12 MSBs des Sendeworts sollen eine Konstante sein
  -- Die 4 LSBs des Sendeworts sollen eine Variable sein und den Wortzaehler
  -- enthalten
  constant SendeKonstante: std_logic_vector(0 to (BitbreiteBus1-4-1)):=
  -- BitbreiteBus1 hat 16 Bit
  -- -4 deswegen, weil eine 12-Bit Konstante deklariert werden soll
  -- -1 deswegen, weil von 0 an gezaehlt wird
   conv_std_logic_vector(16#ABC#, 12);
  -- deklariere 12 Bit-Konstantenvektor
  -- Daran wird ein laufender 4-Bit Wortzaehler angehaengt, so dass sich insgesamt
  -- ein 16-Bit Sendewort ergibt
  -- => Wortzaehler muss eine 4-Bit Groesse sein
  -- Allgemeine Konstanten
  constant active: std_logic:='0'; -- active low logic
  constant inactive: std_logic:='1'; -- active low logic
  -- Liste der Fehler
  type SenderFehlerliste is (KeinFehler, Fehler1); -- enumeration type
  -- Fehler1="Already waited for Inter Frame Gap, but H_TX_DST_RDY_N
  -- is still inactive."
  -- Es wurde bereits auf den IFG gewartet
  -- Konstanten fuer Schleifen
  constant HoechsterWertVonWortzaehler: integer:= 15;
  -- Wortzaehler muss eine 4 Bit-Groesse sein, da er in das letzte Nibble der
  -- Sendekonstante montiert wird
  -- HoechsterWertVonWortzaehler muss mindestens so gross sein, wie
  -- kWorteZumSenden+1;
  -- +1, da sonst der Wortzaehler nach kWorteZumSenden ueberlaueft
  -- Am notwendigen Ausgabeport werden die Fehlermeldungen ausgegeben
  -- Konstanten fuer Takt- und Abschnittsverwaltung
  constant IFGDauer: integer := 6;
  -- IFG hat 6 Takte an Dauer
  constant HoechsterWertVonIFGZaehler : integer := IFGDauer+1;
  -- +1 um zu vermeiden, dass IFGZaehler nach der IFG-Dauer ueberlaeuft
end Konstanten;


-- Package für Prozedur:

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.Konstanten.all;

package Prozeduren is
  procedure EinmaligesSenden(
    variable SendeWort: in std_logic_vector(0 to (BitbreiteBus1-1));
    -- = das Wort, das gesendet werden soll. Muss identisch zur Bitbreite von Bus1
    -- des LLIF sein.
    signal SenderAbschnittszaehler: inout SenderAbschnitte_Typ;
    -- implementiert EA im Hauptprogramm. Wird vom Haupt- und Unterprogramm
    -- geschrieben.
    signal SendePhase: inout SendePhase_Typ;
    -- implementiert EA im Unterprogramm. Wird vom Unterprogramm geschrieben.
    variable IFGZaehler: inout integer range 0 to
```

```
     HoechsterWertVonIFGZaehler;
    -- speichert die Wartezeit fuer den IFG bis zum naechsten Aufruf
    signal Status: out SenderFehlerliste;
    -- liefert den Status des Unterprogramms nach Senden eines Wortes an das
    -- Hauptprogramm zurueck
    -- Aurora LL IF-Signale fuer die Sendeseite
    signal TX_D: out std_logic_vector(0 to BitbreiteBus1-1);
    signal TX_REM: out std_logic;
    signal TX_SOF_N: out std_logic;
    signal TX_EOF_N: out std_logic;
    signal TX_SRC_RDY_N: out std_logic;
    signal TX_DST_RDY_N: in std_logic);
end Prozeduren;

package body Prozeduren is
  procedure EinmaligesSenden(
    variable SendeWort: in std_logic_vector(0 to BitbreiteBus1-1);
    -- = das Wort, das gesendet werden soll. Muss identisch zur Bitbreite von Bus1
    -- des LLIF sein.
    signal SenderAbschnittszaehler: inout SenderAbschnitte_Typ;
    -- implementiert EA im Hauptprogramm. Wird vom Haupt- und Unterprogramm
    -- geschrieben.
    signal SendePhase: inout SendePhase_Typ;
    -- implementiert EA im Unterprogramm. Wird vom Unterprogramm geschrieben.
    variable IFGZaehler: inout integer range 0 to HoechsterWertVonIFGZaehler;
    -- speichert die Wartezeit fuer den IFG bis zum naechsten Aufruf
    signal Status: out SenderFehlerliste;
    -- liefert den Status des Unterprogramms nach Senden eines Wortes an das
    -- Hauptprogramm zurueck
    -- Aurora LL IF-Signale fuer die Sendeseite
    signal TX_D: out std_logic_vector(0 to BitbreiteBus1-1);
    signal TX_REM: out std_logic;
    signal TX_SOF_N: out std_logic;
    signal TX_EOF_N: out std_logic;
    signal TX_SRC_RDY_N: out std_logic;
    signal TX_DST_RDY_N: in std_logic) is
    -- Dieses Unterprogramm benoetigt mehrere Takt zur Ausfuehrung.
    begin
      if SendePhase=PruefenLLIF then
        -- 1. Phase jedes Sendevorgangs
        if TX_DST_RDY_N=active then -- Ist das LL IF bereit?
          -- Ja. Setze einen moeglicherweise erhoehten IFGZaehler
          -- zurueck, da das LL IF nach spaetestens 6 Takten
          -- wieder einsatzbereit geworden ist
          IFGZaehler := 0;
          -- Bereite die Uebergabe von SendeWort an das LL IF vor
          TX_D <= SendeWort;
          -- Bereite die Aktivierung der LL IF-Zeitsignale vor
          TX_REM <= NofRestlicheBytesMinusEins;
          TX_SOF_N <= active;
          TX_EOF_N <= active;
          TX_SRC_RDY_N  <= active;
          -- Register. Werden alle einen Takt spater gueltig
          -- Gehe zur naechsten Phase ueber
          SendePhase <= Einmalige_Datenuebergabe;
          -- Register. Wird einen Takt spater gueltig
          -- Damit wird ein impliziter Taktzaehler realisiert,
          -- und die Zeit von einem Takt fur die Ueberpruefung
          -- des LL IF abgewartet
        else -- DST_RDY_N=inactive
```

```
          -- Das LL IF ist nicht bereit
          -- Warte die Dauer des IFG ab, denn es koennte sich um
          -- einen solchen handeln.
          -- Zaehle IFGDauer an Takten ab
          if IFGZaehler < IFGDauer then
            IFGZaehler := IFGZaehler+1;
            -- Wird einen Takt spaeter gueltig.
            -- Pruefe beim naechsten Programmaufruf, ob das LL IF
            -- bereit oder ob der IFGZaehler uebergelaufen ist.
            -- Alles wird solange gespeichert.
          else -- IFGZaehler >= IFGDauer
            -- IFGDauer wurde abgewartet, trotzdem ist das LL IF
            -- nicht bereit
            Status <= Fehler1;
            -- beende Programm
            SendePhase <= HauptprogrammWeiterschalten;
          end if; -- IFGZaehler <= IFGDauer
        end if; -- TX_DST_RDY_N=active
      elsif SendePhase=Einmalige_Datenuebergabe then
        -- 2. Phase jedes Sendevorgangs
        -- Alle LL IF-Zeitsignale sind jetzt aktiv
        -- Bereite die Deaktivierung der LL IF-Zeitsignale vor
        TX_SOF_N <= inactive;
        TX_EOF_N <= inactive;
        TX_SRC_RDY_N  <= inactive;
        -- Register. Werden einen Takt spaeter gueltig
        -- Gehe zur naechsten Phase ueber
        SendePhase<=SendePause1;
        -- Register. Wird einen Takt spaeter gueltig
        -- Damit wird die Zeit von einem Takt fur die Deaktivierung
        -- der LL IF-Zeitsignale abgewartet
      elsif SendePhase=SendePause1 then
        -- 3. Phase jedes Sendungevorgangs
        -- Damit wird ein Takt fur die Sendepause abgewartet
        Status <= KeinFehler;
        -- es gab keinen Fehler. Gebe dies aus.
        SendePhase <= HauptprogrammWeiterschalten;
        -- Register. Wird einen Takt spater gueltig
      else
        -- SendePhase=HauptprogrammWeiterschalten then
        -- Letzte Phase jedes Sendevorgangs
        -- Damit wird noch ein Takt fur die Sendepause abgewartet
        SendePhase <= PruefenLLIF;
        -- Gehe zurueck an den Anfang und dadurch bereite EA des Unterprogramms
        -- fuer das Senden des naechsten Wortes vor.
        SenderAbschnittszaehler <= StatusPruefen;
        -- Schalte das Hauptprogramms in den naechsten Zustand.
        -- Register. Werden beide einen Takt spater gueltig
      end if; -- SendePhase=PruefenLLIF
    end EinmaligesSenden;
end Prozeduren;


-- SenderEinmaligeDatenuebergabe

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.Konstanten.all;
```

```
use work.Prozeduren.all;

entity SenderEinmaligeDatenuebergabe is
 Port(Drucknopfstart: in std_logic;
      NotwendigerAusgabeport: inout SenderFehlerliste;
      -- H steht fuer Hauaptprogramm
      H_TX_D: out std_logic_vector(0 to BitbreiteBus1-1);
      H_TX_REM: out std_logic;
      H_TX_SOF_N: out std_logic;
      H_TX_EOF_N: out std_logic;
      H_TX_SRC_RDY_N: out std_logic;
      H_TX_DST_RDY_N: in std_logic;
      H_USER_CLK: in STD_lOGIC);
end SenderEinmaligeDatenuebergabe;

architecture One of SenderEinmaligeDatenuebergabe is
  signal H_SenderAbschnittszaehler: SenderAbschnitte_Typ:= LLIFInitialisieren;
  -- SenderAbschnittszaehler fuer das Hauptprogramm. Starte mit
LLIFInitialisieren.
  signal H_SendePhase: SendePhase_Typ:= PruefenLLIF;
  -- SenderAbschnittszaehler fuer das Unterprogramm. Starte mit PruefenLLIF.
  signal H_SenderStatus: SenderFehlerliste := keinFehler;
  -- Register. Initialisiere mit keinFehler.
  begin
   process(H_USER_CLK)
      variable H_SendeWort: std_logic_vector(0 to (BitbreiteBus1-1)):=
(others=>'0');
      -- initialisiere das, was gesendet werden soll, mit Nullen
      variable Wortzaehler: integer range 0 to HoechsterWertVonWortzaehler := 0;
      -- Wird verwendet, um die Worte zu zaehlen. Beginne mit 0.
      variable H_IFGZaehler: integer range 0 to HoechsterWertVonIFGZaehler := 0;
      -- Wird verwendet, um die Zeit beim IFG abzuwarten.
      -- Beginne mit 0.
      begin
        if H_USER_CLK'event and H_USER_CLK = '1' then
          -- starte mit jedem ansteigendem Takt von H_USER_CLK neu
          if H_SenderAbschnittszaehler = LLIFInitialisieren then
            -- 1 Takt Zeit fuer LL IF-Initialisierung
            -- Bereite die Deaktivierung der LL IF-Zeitsignale vor
            H_TX_SOF_N <= inactive;
            H_TX_EOF_N <= inactive;
            H_TX_SRC_RDY_N  <= inactive;
            -- Register. Werden einen Takt spaeter gueltig
            -- Gehe zur naechsten Phase ueber
            H_SenderAbschnittszaehler <= DrucknopfAbwarten;
          elsif H_SenderAbschnittszaehler = DrucknopfAbwarten then
            if Drucknopfstart='1' then
              -- Chipscope ist bereit. Dies ist die einzige Moeglichkeit, die
              -- Schleife zu verlassen
              H_SenderAbschnittszaehler <= EinWortSenden;
            else -- Drucknopfstart/='1'
              null; -- tue nichts, bleibe in DrucknopfAbwarten
              -- LL IF-Zeitsignale bleiben wie in der LL IF-Initialisierung
              -- teste beim nachsten Aufruf, ob Druckknopf gedrueckt ist
            end if; -- Drucknopfstart='1'
          elsif H_SenderAbschnittszaehler = EinWortSenden then
            -- Hauptprogramm ist im Sendemodus
            -- bereite das H_SendeWort vor
            H_SendeWort := SendeKonstante & conv_std_logic_vector(Wortzaehler, 4);
            -- SendeKonstante ist ein 12 Bit Konstante und dient als 12 MSBs
```

```
            -- Die 4 LSBs des H_SendeWorts enthalten den Wortzaehler.
            -- Beides wird aneinandergehaengt
            -- Kein Register. Wird sofort gueltig.
            -- Rufe die einmalige Datenuebergabe auf.
            EinmaligesSenden(H_SendeWort, H_SenderAbschnittszaehler, H_SendePhase,
            H_IFGZaehler, H_SenderStatus, H_TX_D, H_TX_REM, H_TX_SOF_N,
            H_TX_EOF_N, H_TX_SRC_RDY_N, H_TX_DST_RDY_N);
            -- Achtung: das Unterprogramm wird mit jeder Taktflanke erneut
aufgerufen
            -- Es muss deshalb reentrant sein und selber das Weiterschalten in den
            -- naechsten Abschnitt vornehmen.
          elsif H_SenderAbschnittszaehler = StatusPruefen then
            -- 1 Wort wurde möglicherweise erfolgreich gesendet
            -- H_SenderStatus enthält den Fehlerstatus.
            -- Gebe den Status am notwendigen Ausgabeport aus
            NotwendigerAusgabeport <= H_SenderStatus;
            -- Register. Wird einen Takt spaeter gueltig
            -- Pruefe den Fehlerstatus
            if H_SenderStatus /= KeinFehler then
              H_SenderAbschnittszaehler <= Stop;
              -- Register. Wird einen Takt spaeter gueltig
            else -- H_SenderStatus = KeinFehler
              -- 1 Wort wurde erfolgreich gesendet
              -- Erhoehe Wortzaehler
              Wortzaehler := Wortzaehler+1;
              -- Register. Wird einen Takt spaeter gueltig
              -- noch nicht kWorteZumSenden gesendet?
              if Wortzaehler<kWorteZumSenden then
                H_SenderAbschnittszaehler <= EinWortSenden;
                -- Sende noch 1 Wort.
                -- Register. Wird einen Takt spaeter gueltig
              else --Wortzaehler>=kWorteZumSenden
                -- alle Wort wurden gesendet
                H_SenderAbschnittszaehler <= Stop;
                -- Register. Wird einen Takt spaeter gueltig
              end if; -- Wortzaehler<kWorteZumSenden
            end if; -- H_SenderStatus /= KeinFehler
          else
            -- H_SenderAbschnittszaehler = Stop
            -- null;
            -- Beende Programm
          end if; -- H_SenderAbschnittszaehler = DrucknopfAbwarten
        end if; -- H_USER_CLK'event and H_USER_CLK = '1' ohne else
      end process;
   end One;
```

Empfängerseite:

```
- Dateiname: Einmalige Datenuebergabe S u E V14.fm
-- Autor: HRI 16.02.11
-- Titel: Empfaenger L1c von CarRing II
-- Beschreibung:
-- Dieses Programm bezieht sich auf das Dokument:
-- H. Richter, Beschreibung von Local Link, HR 08.11.10
-- Es wird der Fall "Empfangen weniger Daten (einmalige Datenuebergabe)"
-- implementiert. Dazu muss dieser Code vor dem Sendercode in das
-- empfangende FPGA geladen und gestartet werden.
-- In einer endlichen Schleife werden 3 Worte der Wortbreite n=2 Byte vom
-- LL IF (= L1b im ISO-Modell) uebernommen
```

```
-- Die Zahl z der dazu benoetigten Zyklen von USER_CLK auf Bus 3 ist 3*5.
-- Die Empfaengerseite des LL IF kann jederzeit ueber das
-- Signal RX_SRC_RDY_N anzeigen, dass eine Betriebsunterbrechung vorliegt.
-- Der Empfaenger hat stabile Ausgangssignale mit jeder fallenden Flanke von
-- USER_CLK. Deshalb wird der Empfaenger-Code mit jeder fallenden Flanke aktiviert.
-- Ist fuer eine maximale Empfangsgeschwindigkeit von 4 Takten pro Wort ausgelegt.


-- Package für Konstanten:

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package Konstanten_Empf is

  type EmpfaengerAbschnitte_Typ is (LLIFInitialisieren, EinWortEmpfangen,
   StatusPruefen, Stop);
  -- Hauptprogramm hat 3 Abschnitt + Stop
  -- Unterprogramm hat nur einen Abschnitt und keinen EA
  -- Allgemeine Konstanten
  constant active: std_logic:='0'; -- active low logic
  constant inactive: std_logic:='1'; -- active low logic
  -- Liste der Fehler
  type EmpfaengerFehlerliste is (KeinFehler, Fehler1, Fehler2); -- enumeration type
  -- Fehler1 = RX_EOF_N Inaktiv, Aktiv Erwartet
  -- Fehler2 = RX_REM_UngleicNofRestlicheBytesMinusEins
  -- Konstanten fuer das LL IF
  constant kWorteZumEmpfangen: integer:= 3; -- empfange k=3 Worte
  -- Das, was empfangen werden soll, sind 2 Byte-Worte
  constant WortbreiteBus3: integer:= 2;
  --Bus 3 hat die Wortbreite n=2 gemaess S. 24 von UG224.pdf
  constant BitbreiteBus3: integer:= WortbreiteBus3*8;
  --Bus 3 hat 16 Bit Wortbreite
  --Bus 4 hat 1 Bit Wortbreite und kann den Wert 0 oder 1 vom LL IF anliefern
  constant NofRestlicheBytesMinusEins: std_logic:='1';
  -- Der Rest, der auf Bus 4 angegeben wird, ist NofRestlicheBytesMinusEins,
  -- d.h. das maximal Moegliche
  constant HoechsterWertVonWortzaehler: integer:= 16#FF#;
  -- Wortzaehler muss eine 4 Bit-Groesse sein, da er im letzten Nibble des
  -- Empfangswortes steht
  -- HoechsterWertVonWortzaehler muss mindestens so gross sein, wie
  -- kWorteZumEmpfangen+1;
  -- +1, da sonst der Wortzaehler nach kWorteZumEmpfangen ueberlaueft
  constant NiedrigsterWertVonEmpfangsWort: integer := 0;
  -- 16 bit Integer. Muss die gleiche Bitzahl wie Bus 3 haben
  constant HoechsterWertVonEmpfangsWort: integer := (2**BitbreiteBus3-1);
  -- 16 bit Integer. Muss die gleiche Bitzahl wie Bus 3 haben
  constant NiedrigsterWertAmAusgabeport: integer :=
NiedrigsterWertVonEmpfangsWort;
  constant HoechsterWertAmAusgabeport: integer := HoechsterWertVonEmpfangsWort;
  -- Ausgabeport gibt das EmpfangsWort aus

end Konstanten_Empf;


-- Package für Prozedur:

library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.Konstanten_Empf.all;

package Prozeduren_Empf is
  procedure EinmaligesEmpfangen(
    variable EmpfangsWort: out integer range 0 to HoechsterWertVonEmpfangsWort;
    signal EmpfaengerAbschnittszaehler: inout EmpfaengerAbschnitte_Typ;
    -- implementiert EA im Hauptprogramm. Wird vom Haupt- und Unterprogramm
    -- geschrieben. Ist ein EA mit Ein- und Ausgabe, der die einmalige
    -- Datenuebergabe gemaess des im Eingang erwähnten Dokuments abarbeitet.
    -- Eingabe in den EA ist USER_CLK sowie alle Ausgaben des LL IF, d.h.
    -- H_RX_D, H_RX_REM, H_RX_SOF_N, H_RX_EOF_N und H_RX_SRC_RDY_N
    signal Status: inout EmpfaengerFehlerliste;
    -- liefert den Status des Empfaengers zurueck.
    -- Aurora LL IF-Signale auf der Empfaengerseite:
    signal RX_D: in std_logic_vector(0 to BitbreiteBus3-1);
    signal RX_REM: in std_logic;
    signal RX_SOF_N: in std_logic;
    signal RX_EOF_N: in std_logic;
    signal RX_SRC_RDY_N: in std_logic;
    signal RX_DST_RDY_N: out std_logic);
end Prozeduren_Empf;

package body Prozeduren_Empf is
  procedure EinmaligesEmpfangen(
    variable EmpfangsWort: out integer range 0 to HoechsterWertVonEmpfangsWort;
    signal EmpfaengerAbschnittszaehler: inout EmpfaengerAbschnitte_Typ;
    -- implementiert EA im Hauptprogramm. Wird vom Haupt- und Unterprogramm
    -- geschrieben.
    signal Status: inout EmpfaengerFehlerliste;
    -- liefert den Status des Empfaengers zurueck.
    -- Aurora LL IF-Signale auf der Empfaengerseite:
    signal RX_D: in std_logic_vector(0 to BitbreiteBus3-1);
    signal RX_REM: in std_logic;
    signal RX_SOF_N: in std_logic;
    signal RX_EOF_N: in std_logic;
    signal RX_SRC_RDY_N: in std_logic;
    signal RX_DST_RDY_N: out std_logic) is
  begin
    -- Achtung: rufendes Programm von EinmaligesEmpfangen muss mit
    -- fallender Flanke von USER_CLK aufrufen, da dann die Signale am
    -- LL IF stabil anliegen
    -- Dieses Unterprogramm kann mehrere Takt zur Ausfuehrung benoetigen.
    -- Sind die Signale am LL IF gueltig?
    if RX_SRC_RDY_N=active then
      -- Pruefe, ob Daten am LL IF anliegen
      -- Achtung: es gibt keinen Empfaenger-Timeout. Diese Stelle kann
      -- beliebig oft durchlaufen werden. Gefahr der Endlosschleife.
      if RX_SOF_N = active then
      -- Achtung: es gibt keinen Empfaenger-Timeout. Diese Stelle kann
      -- ebenfalls beliebig oft durchlaufen werden. Gefahr der
      -- Endlosschleife.
        -- Es liegen Daten an
        EmpfangsWort := conv_integer(RX_D);
        -- Retten der Daten
        -- Schreiben vor dem Lesen. Kein Register.
        -- Wird sofort gueltig.
        -- Status ausgeben. Pruefen, ob RX_EOF_N = active ist
```

```
        if RX_EOF_N = active then
          -- RX_EOF_N ist O.K.
          null -- tue nichts
        else -- RX_EOF_N = inactive
          -- Fehlerfall. Es kann sich nicht um eine einmalige
          -- Datenuebertragung handeln
          Status <= Fehler1;
        end if; -- RX_EOF_N = active
        -- Pruefen, ob RX_REM richtig steht
        if RX_REM = NofRestlicheBytesMinusEins then
          -- RX_REM ist O.K.
          null -- tue nichts
        else -- RX_REM ungleich NofRestlicheBytesMinusEins
          -- Fehlerfall, da Anzahl Bytes auf Bus 4 falsch ist
          Status <= Fehler2;
          -- Falls Fehler2 auftritt, ist er hochprior und überschreibt Fehler1.
          -- Fehler2 kann nur gesetzt werden, wenn RX_REM nicht als Konstante
          -- auf NofRestlicheBytesMinusEins im rufenden Programm, d.h. in Aurora
          -- gesetzt worden ist. Andernfalls bleibt das Bit, das Fehler2 kodiert
          -- nicht angeschlossen und wird vom Compiler wegoptimiert.
        end if; -- RX_REM = NofRestlicheBytesMinusEins
        if (Status /= Fehler1) and (Status /= Fehler2)
          -- es gab keinen Fehler. Gebe dies aus.
          Status <= KeinFehler;
        end if; -- if ohne else für Status /= (Fehler1 or Fehler2)
        -- Schalte Hauptprogramm weiter.
        EmpfaengerAbschnittszaehler <= StatusPruefen;
        -- Register. Wird einen Takt spater gueltig
      end if; -- RX_SOF_N = active
    end if; -- RX_SRC_RDY_N=active
  end EinmaligesEmpfangen;
end Prozeduren_Empf;

-- EmpfaengerEinmaligeDatenuebergabe

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.Konstanten_Empf.all;
use work.Prozeduren_Empf.all;

entity EmpfaengerEinmaligeDatenuebergabe is
  Port(NotwendigerAusgabeport: inout integer
        range NiedrigsterWertAmAusgabeport to HoechsterWertAmAusgabeport;
       -- H steht fuer Hauptprogramm
       H_RX_D: in std_logic_vector(0 to BitbreiteBus3-1);
       H_RX_REM: in std_logic;
       H_RX_SOF_N: in std_logic;
       H_RX_EOF_N: in std_logic;
       H_RX_SRC_RDY_N: in std_logic;
       H_RX_DST_RDY_N: out std_logic;
       H_USER_CLK: in STD_lOGIC);
end EmpfaengerEinmaligeDatenuebergabe;

architecture One of EmpfaengerEinmaligeDatenuebergabe is
  signal H_EmpfaengerAbschnittszaehler: EmpfaengerAbschnitte_Typ:=
   LLIFInitialisieren;
  -- EmpfaengerAbschnittszaehler fuer das Hauptprogramm. Starte mit
  -- LLIFInitialisieren.
```

```
  signal H_EmpfaengerStatus: EmpfaengerFehlerliste := keinFehler;
  -- Register. Initialisiere mit keinFehler.
  begin
    process(H_USER_CLK)
      variable Wortzaehler: integer
       range 0 to HoechsterWertVonWortzaehler := 0;
      -- Wird verwendet, um die Worte zu zaehlen. Beginne mit Wort 0.
      variable H_EmpfangsWort: integer range 0 to HoechsterWertVonEmpfangsWort :=
0;
      -- Initialisiere mit Wort 0.
      begin
        if H_USER_CLK'event and H_USER_CLK = '0' then
          -- starte mit jedem fallenden Takt von H_USER_CLK neu, da Daten mit
          -- fallender Flanke von H_USER_CLK an der Empfaengerseite des LL IF
          -- stabil anliegen.
          if H_EmpfaengerAbschnittszaehler = LLIFInitialisieren then
            H_RX_DST_RDY_N <= inaktiv;
            -- Empfaenger ist nicht bereit, da in der Initialisierungsphase
            -- Register. Wird einen Takt spaeter gueltig.
            H_EmpfaengerAbschnittszaehler <= EinWortEmpfangen;
            -- 1 Takt Zeit fuer LL IF-Initialisierung
          elsif H_EmpfaengerAbschnittszaehler = EinWortEmpfangen then
            H_RX_DST_RDY_N <= aktiv;
            -- Empfaenger ist jetzt bereit und wartet darauf, dass ein
            -- Datenrahmen empfangen wird
            -- Register. Wird einen Takt spaeter gueltig.
            -- 1 Wort soll empfangen werden
            EinmaligesEmpfangen(H_EmpfangsWort, H_EmpfaengerAbschnittszaehler,
             H_EmpfaengerStatus, H_RX_D, H_RX_REM, H_RX_SOF_N, H_RX_EOF_N,
             H_RX_SRC_RDY_N, H_RX_DST_RDY_N);
            -- Achtung: das Unterprogramm wird mit jeder Taktflanke erneut
aufgerufen
            -- Es muss deshalb reentrant sein und selber das Weiterschalten in den
            -- naechsten Abschnitt vornehmen.
          elsif H_EmpfaengerAbschnittszaehler = StatusPruefen then
            -- H_EmpfaengerStatus enthält den Fehlerstatus.
            -- Pruefe den Fehlerstatus
            if H_EmpfaengerStatus /= KeinFehler then
              H_EmpfaengerAbschnittszaehler <= Stop;
              -- Register. Wird einen Takt spaeter gueltig
            else -- H_EmpfaengerStatus = KeinFehler
              -- 1 Wort wurde erfolgreich empfangen
              -- Gebe das Empfangswort am notwendigen Ausgabeport aus
              NotwendigerAusgabeport <= H_EmpfangsWort;
              -- Register. Wird einen Takt spaeter gueltig
              -- Erhoehe Wortzaehler
              Wortzaehler := Wortzaehler+1;
              -- Register. Wird einen Takt spaeter gueltig
              -- noch nicht alle Worte empfangen?
              if Wortzaehler<kWorteZumEmpfangen then
                H_EmpfaengerAbschnittszaehler <= EinWortEmpfangen;
                -- Empfange noch 1 Wort.
                -- Register. Wird einen Takt spaeter gueltig
              else --Wortzaehler>=kWorteZumEmpfangen
                -- alle Wort wurden empfangen
                H_EmpfaengerAbschnittszaehler <= Stop;
                -- Register. Wird einen Takt spaeter gueltig
              end if; -- Wortzaehler<kWorteZumEmpfangen
            end if; -- H_EmpfaengerStatus /= KeinFehler
          else
```

```
          -- H_EmpfaengerAbschnittszaehler = Stop
          null;
          -- Beende Programm
        end if; -- H_Abschnittszaehler = LLIFInitialisieren
      end if; -- H_USER_CLK'event and H_USER_CLK = '0' ohne else
    end process;
  end One;
```

Programmcode 26: Teil der Schicht 1c von CarRingII.

Beachtenswert ist in Code 26 u.a. die Weiterschaltung des Programmcodes durch gekoppelte endliche Automaten, wie sie in Kapitel 7.3.2 "Realisierung durch gekoppelte endliche Automaten" beschrieben wurde, und die Handhabung der Statusbits des Local Link Interfaces, die im Kapitel 6.14 "Mehrfache Zuweisungen an Signale und Variable mit if" erläutert ist.

14 Praktische Hinweise zur VHDL-Programmierung

14.1 Simulation und Chip-Synthese

VHDL-Programme können auf zwei grundsätzlich verschiedene Arten ausgeführt werden:

1.) in einem Software-Simulator wie z.B. Modelsim von Mentor Graphics
2.) in einem FPGA wie z.B. Spartan 3 oder Virtex 6 von Xilinx

Der normale Weg bei der VHDL- und Verilog-Programmentwicklung ist, dass das Programm zuerst simulativ ausgeführt und danach die Chip-Synthese begonnen wird, sofern der Simulator das Programm fehlerfrei übersetzen und ausführen konnte, und ein ausgiebiger Funktionstest erfolgreich war. Bei der Ausführung im Simulator wird stets der gesamte Sprachumfang von VHDL unterstützt, meistens sogar in der aktuellen Sprachdefinition, und es können -im Prinzip- beliebig lange und komplexe Programme simuliert werden. Dies ist bei den Synthesewerkzeugen leider nicht der Fall.
Die simulative Abarbeitung der Befehle soll den Programmentwurf wirksam unterstützen und bereits vor der Chip-Synthese auf Fehler hinweisen. Dazu gehorcht jeder VHDL-Simulator denselben Regeln, die seine Effizienz bei der Simulation erhöhen, und die zugleich die spätere Ausführung auf dem Chip nachahmen sollen. Da diese beiden Zielsetzungen verschieden sind, kommt es vor, dass die Simulation und die FPGA-Ausführung voneinander abweichen. Die Abweichungen sind in beiden Richtungen möglich, d.h. es gibt Programme, die sich simulieren aber nicht synthetisieren lassen und ungekehrt. Häufig ist das erste, das zweite ist die Ausnahme. Die Regeln, nach denen alle VHDL-Simulatoren arbeiten, sind:

1.) Prozesse werden nur aktiviert, wenn sich in der sensitivity list des Prozesses etwas ändert. Diese Vorgehensweise wird als diskrete Ereignissimulation bezeichnet und ist der Grund, warum es die sensitivity list gibt. Diese Liste enthält alle potentiellen Triggersignale, die den Prozess aktivieren können und sorgt für hohe Effizienz bei der Ausführung der Simulation, da ein Simulationslauf erst nach Veränderung einer Eingabegröße durchgeführt wird. Es wird also nicht ununterbrochen simuliert. Handelt es sich um die Simulation taktsynchroner Prozesse, wird jeder Prozess mit der Änderung des Taktsignals aktiviert, der dazu

in der sensitivity list aufgeführt sein muss. Die Prozessaktivierung geschieht häufig durch eine ansteigende Taktflanke mittels Angabe von `if clk'event and clk='1'` oder `wait until clk'event and clk='1'`.

2.) Zuweisungen an Signale werden vom Simulator gesammelt und erst am Ende des Prozesses ausgeführt, so dass deren Ergebnisse bei der nächsten Prozessausführung zur Verfügung stehen. Dies ahmt die Tatsache nach, dass Signale stets als speichernde Elemente wie Register, Zähler, Akkumulator oder Schieberegister synthetisiert werden, und dass solche Elemente ihren Eingangswert in der Regel erst mit der nächsten Taktflanke speichern können. Die nächste Taktflanke aktiviert zugleich den nächsten Simulationsdurchlauf, so dass mit Beginn der nächsten Prozessaktivierung auch das Registerausgangssignal zur Verfügung steht. Dies ahmt die Wirklichkeit gut nach, vorausgesetzt, dass auf dem Chip tatsächlich speichernde Elemente verwendet werden, die wie D-Flip Flops angesteuert werden und nicht wie Latches. Auf die Verwendung von Latches, die nicht taktflanken- sondern taktpegelgesteuert sind, sollte auf dem FPGA deshalb verzichtet werden, wenn die Simulation genau sein soll.

3.) Von mehreren Zuweisungen an dasselbe Signal im selben Prozess wird nur die letzte ausgeführt. D.h. vorangehende Zuweisungen an ein Signal werden durch die letzte Zuweisung überschrieben. Dies entspricht der Tatsache, dass speichernde Elemente in der Regel den Eingangswert mit einer Taktflanke übernehmen. Signale, die vor einer Taktflanke anliegen, werden nicht übernommen, vorausgesetzt sind taktflanken- nicht taktpegelgesteuert. Bei dieser Vorgehensweise wird die Wirklichkeit allerdings nur näherungsweise nachgebildet, da die setup- und hold-Zeiten der speichernde Elemente vernachlässigt werden.

4.) Dementsprechend werden Zuweisungen an Variable nur dann sofort ausgeführt, wenn sie nicht als speichernde Elemente synthetisiert werden. Solche Variable sind reine Compiler-interne Zwischengrößen. Werden Variable hingegen genau wie Signale als speichernde Elemente simuliert, stehen deren Ergebnisse erst bei der nächsten Prozessaktivierung zur Verfügung. Dies ahmt die Wirklichkeit gut nach, denn Variable, die kein speicherndes Elemente zur Folge haben, werden auf dem Chip nur als Verbindung (=“Draht“) oder als Bündel von Verbindungen synthetisiert. Eine binäre Variable wird durch eine einzige Verbindung repräsentiert, alle mehrstelligen Variablen durch ein Bündel von Verbindungen.

14.2 Endliche Automaten mit Ein- und Ausgabe

Endliche Automaten (EAs) werden im hardware-orientierten und im Software-orientierten Programmierstil eingesetzt. Sie erfordern stets einen Zustandsspeicher, d.h. ein Register, das den Zustand bis zum nächsten Taktschlag zwischenspeichert. Es wird davon abgeraten, den Zustandsspeicher als Variablenregister zu implementieren, da das Variablenregister verschwindet, sobald in allen Automatenzuständen eine Zuweisung an die Zustandsvariable erfolgt, was in der Regel bei EAs der Fall ist. D.h, EAs sind mit einem Signal als Zustandsspeicher zu implementieren.

Ferner sind die Zustände des EAs als die Elemente eines VHDL Enumeration Types zu spezifizieren und nicht als explizite Zahlenwerte. Der Grund, warum es zu vermeiden ist, dem Zustandssignal des EAs explizite Zahlenwerte vorzugeben, ist in Kapitel 6.14 "Mehrfache Zuweisungen an Signale und Variable mit if" erläutert. Im Falle der Spezifikation der EA-

Zustände über einen Enumeration Types wählt das Synthesewerkzeug selbstständig aus der Menge der möglichen Kodierungsarten für Enumeration Types eine Art aus. Typische Kodierungsarten für Enumeration Types sind u.a.:

1.) Die Elemente des VHDL Enumeration types werden in der Aufschreibreihenfolge kodiert (= sequential encoding).
2.) Die Elemente des VHDL Enumeration types werden im Gray Code kodiert (= Gray encoding.
3.) Die Elemente des VHDL Enumeration types werden so kodiert, dass jeder Zustand durch ein Statusbit repräsentiert wird (= one hot encoding).

Welche Kodierungsart das Synthesewerkzeug wählt, ist herstellerspezifisch. Man kann eine bestimmte Kodierungsart erzwingen, indem man ein sog. Attribut in den VHDL Code einfügt. Leider sind VHDL-Attribute ebenfalls herstellerspezifisch, so dass davon abgeraten wird sie häufig zu verwenden, weil der dann Code nicht mehr von Synthesewerkzeugen verschiedener Hersteller unverändert übersetzt werden kann.
Es gibt allerdings eine Situation, in der es unvermeidlich ist, ein Attribut einzusetzen: Sobald ein EA bestehend aus mehreren Flip Flops einen EA-Eingang auf High- oder Low-Pegel beobachtet, muss unbedingt die Cray-Kodierung verwendet werden, falls der Eingang sich asynchron zum Takt des EAs ändern kann. Andernfalls werden die Flip Flops im Laufe der Zeit mit einer zwar geringen aber von Null verschiedenen Wahrscheinlichkeit unterschiedliche Urteile darüber abgeben, ob der Eingang schon High oder noch Low ist. Das Resultat sind sporadische Zustandsübergänge im EA, die im VHDL-Programm nicht definiert sind und demzufolge große Verwirrung beim Programmierer auslösen. Bei der Cray-Kodierung ist eine unterschiedliche Bewertung durch verschiedene Flip Flops ausgeschlossen, weil im jedem Zustand nur ein Flip Flop existiert, das über das Weiterschalten in einen Nachfolgezustand entscheidet. Für eine Analyse und Fehlersuche des so synthetisierten EAs ist der Synthesebericht zu konsultieren. Daraus kann entnommen werden, welcher EA-Zustand mit welchem numerischen Zahlenwert korreliert.

14.3 Parameterübergabe in Prozeduren und Funktionen

In allen prozeduralen Programmiersprachen werden von einem rufenden Programm aus Parameter in einer geordneten Liste an Prozeduren und Funktionen übergeben. Die Aufrufparameter des rufenden Programms werden auch als aktuelle Parameter bezeichnet, und die des gerufenen Programms als formale Parameter. In Synplify und xst ist es möglich, den aktuellen und den formalen Parametern denselben Namen zu geben, ohne dass es eine Fehlermeldung gibt. Davon Gebrauch zu machen wird aus mehreren Gründen abgeraten, auch wenn es als Vereinfachung erscheint. Zum einen verschlechtert sich die Qualität des Syntheseberichts, da Variable und Signale gleichzeitig in zwei verschiedenen Zusammenhängen (Haupt- und Unterprogramm) benutzt werden. Zum andereren kann das Synthesewerkzeug u.U. eine falsche Netzliste erzeugen, ohne dass ein Warning ausgegeben wird.

14.4 ISE, xst und Synplify

Für den Technical Report wurden Xilinx ISE (=Integrated Software Environment), Xilinx xst = Xilinx Synthese Tool) und Synopsis Synplify Premier als Synthesewerkzeuge verwendet. Im weiteren werden Hinweise für den Umgang mit diesen Werkzeugen gegeben, die aus dem täglichen Umgang damit gewonnen wurden.

1.) Synplify Premier ist wie xst nur ein Synthesewerkzeug und keine Entwicklungsumgebung zur VHDL-Programmierung. Um Synplify Premier sinnvoll einzusetzen, muss die ISE-Entwicklungsumgebung verwendet und dabei xst durch Synplify ersetzt werden. Dies ist in ISE möglich, da in diese Umgebung auch das Synthesewerkzeug eines anderen Herstellers eingebettet werden kann.
2.) Bei Synplify Premier gibt es die beiden Varianten der logischen und der physikalischen Synthese. Diese unterscheiden sich darin, ob nur eine Netzliste erstellt oder ob eine grobe Komponentenplatzierung auf dem Chip bereits durch Synplify erfolgt. Wird die logische Synthese gewählt, übernimmt ISE die komplette Komponentenplatzierung. Bei der physikalischen Synthese führt ISE die Feinplatzierung und Optimierung durch. Wenn das Syntheseergebnis hoch optimiert sein soll, dann sollten beide Varianten getestet werden.
3.) Es ist sinnvoll, sich den vom Synthesewerkzeug erzeugten Schaltplan (Schematic Diagram) anzusehen und Bezüge zwischen der generierten Hardware und der eigenen Software herzustellen. Dies fördert erheblich das Verständnis dafür, was das Synthesewerkzeug macht.
4.) Bei der VHDL-Programmübersetzung kann es zu sog. Warning-Meldungen im Syntheseberichte kommen. Für jedes Warning existiert eine Ursache. Diese muss, sofern irgend möglich, beseitigt werden, auch wenn der Compiler trotz eines Warnings den Code übersetzt. D.h., ein Warning sollte in der Regel nicht ignoriert werden, um verdeckte Probleme und Programmierfehler aufzudecken und zu beheben.
5.) Xst und Synplify bewerten kleinere Fehler (Warnings), die es in VDHL-Programmen gibt, unterschiedlich. D.h. das eine Synthesewerkzeug entdeckt ein Problem, das das andere nicht findet und umgekehrt. Um einen möglichst grossen Informationsgewinn bei der Synthese zu erhalten, ist es deshalb empfehlenswert, dasselbe Programm sowohl mit xst als auch mit Synplify zu übersetzen und sich in beiden Syntheseberichten alle Fehlermeldungen, d.h. auch Warning-Meldungen genau anzusehen.
6.) Die Verwendung von „fprint“ aus dem öffentlich erhältlichen print package work.PCK_FIO.all ist zum Debuggen von VHDL-Programmen in der Simulationsphase gut geeignet. Die Ausführung von fprint verbraucht in der Simulation 0 Takte Ausführungszeit. D.h. es können beliebig viele fprints in einem Prozess ausgeführt werden.
7.) Manche Syntheseprogramme (xst) benötigen mindestens einen Ausgabeport, damit sie einen sinnvollen Schaltplan erzeugen. In der vorliegenden Darstellung haben deshalb alle Beispielprogramme einen Ausgabeport, der den Namen „Notwendiger-Ausgabeport“ trägt.
8.) Die Chip-Synthese ist derzeit noch nicht perfekt. Das bedeutet, dass in der Regel im Vergleich zur Simulation nur weniger komplexe und weniger lange Programme erfolgreich synthetisiert werden können. Insgesamt ist die Chip-Synthese ein nicht-trivialer Vorgang mit nicht-linearer Zeitkomplexität. Sie kann hohe Rechenzeit und viel Speicher erfordern.

Laufzeiten, die länger als 30 Minuten dauern und mehr als 4 GByte an Speicher brauchen, deuten aber oft auf einen Programmierfehler hin und sollten abgebrochen werden.

9.) Synthesewerkzeuge haben darüberhinaus die Eigenschaft, dass sie keine Fehlermeldung ausgeben, wenn sie Teile des Programms nicht synthetisieren können. Schwierige, vom Synthesewerkzeuge nicht verstandene Code-Teile werden kommentarlos ignoriert. Bei xst kommt als weitere Schwierigkeit noch hinzu, dass das Studium des von xst erzeugten Schaltplans in dieser Situation nicht weiter hilft, weil xst-Schaltpläne unvollständig sein können, d.h. sie können weniger Leitungen enthalten als tatsächlich synthetisiert wurden. Nur der Compiler Report ist vollständig. Das macht die Chip-Synthese schwierig. Bei größeren Programmen kann eine solche Situation nur durch aufwendiges Debuggen gefunden werden.

10.)Die Anzeige von Variablen und Signalen bei Chipscope kann Bit, Integer in Dezimaldarstellung, Integer in Hexadezimaldarstellung und andere Anzeigetypen umfassen. Bei allen VHDL-Datentypen, die mehr als ein Bit umfassen, ist es übersichtlicher, die Darstellung als Bitvektor tatsächlich nur für Bitvektoren einzusetzen und für die Anzeige der anderen Datentypen die Anzeigemöglichkeiten die Chipscope bietet, auch einzusetzen.

11.)Im Synthesebericht von xst wird bei der Auflistung der vom Synthesewerkzeug gewählten Codierung der Zustände endlicher Automaten eine verdrehte Bitreihenfolge verwendet, in der das LSB links und das MSB rechts steht.

14.5 Chipscope

Chipscope ist ein wichtiges Software-Werkzeug zur Erstellung und Visualisierung von sog. post mortem dumps, das unbedingt als Testhilfe sowohl für xst als auch für Synplify Premier verwendet werden sollte. Der Begriff post mortem dump stammt ursprünglich aus der Rechnerprogrammierung und bezeichnet Speicherauszüge, die der Prozessor während der Programmausführung in einem eigens dafür reservierten Speicherbereich zum Zwecke des Programmtestens abgelegt hat. Nach Beendigung des Programms wurde der Speicherauszug auf die Festplatte transferiert und anschließend entweder als Text, Zahlenkolonnen oder graphisch dargestellt. Ganz ähnlich arbeitet Chipscope. Darüberhinaus ist eine graphische Darstellung im Stile eines Logikanalysators gegeben. Die Speicherauszüge (post mortem dumps) werden manchmal auch als log (=Logbuch) bezeichnet. Damit überhaupt Speicherauszüge geschrieben werden können, muss auf dem FPGA on-chip RAM oder auf der Platine, auf der sich das FPGA befindet, off-chip RAM vorhanden sein. Zusätzlich muss das zu testende Programm „instrumentiert“, d.h. mit zusätzlichem Test-Code versehen werden, der das Schreiben in den reservierten Speicherbereich vornimmt. Instrumentieren heißt, dass der Programmierer die Programmteile, die für Chipscope auf dem Chip vorhanden sein müssen (ICON und ILA), in den eigenen VHDL-Code integrieren muss. Nach dem Programm-Download sammelt ICON von bis zu 15 ILA Cores die Daten ein und kommuniziert mit der Chipscope Software (Chipscope Pro) auf dem PC. Im Unterschied zu ICON und ILA werden bei einer Variante von Chipscope, dem VIO Core, die geschriebenen Werte bereits während der Programmausführung zum Programmierrechner (Host) übertragen und dort dargestellt. Für den Upload der logging-Dateien bei ICON und ILA bzw. der Echtzeit-Daten bei VIO wird wie beim Programmladen die JTAG-Schnittstelle des FPGAs verwendet.

Um zum richtigen Zeitpunkt mit der Aufzeichnung von Kontrollausdrucken beginnen zu können, kann ein internes oder ein externes Triggersignal werden. Beim Triggern der Datenaufnahme ist darauf zu achten, dass das Empfängerteil von Chipscope (Chipscope Pro), das im PC angesiedelt ist und das über die JTAG-Programmierschnittstelle den Speicherauszug vom FPGA in Empfang nimmt, startbereit ist, bevor das VHDL-Programm vom FPGA ausgeführt wird, sonst gehen Daten zu Beginn der Programmausführung verloren. Das kann man z.B, dadurch erreichen, dass zu Beginn der Programmausführung vom FPGA abgefragt wird, ob ein Druckknopf gedrückt ist.

14.6 Modelsim

Modelsim ist ein de-facto Standard für die simulative Ausführung von VHDL-Programmen. Es unterstützt vollständig den aktuellen VHDL Sprachumfang, der in definiert ist. Es wird als unterstützendes Testhilfsmittel empfohlen, allerdings ist jeder VHDL-Simulator zum Test von synthesefähigem Code nur bedingt geeignet, da, wie bereits erwähnt, ein großer Unterschied zwischen dem existiert, was in VDHL programmiert und simuliert werden kann und zwischen dem, was tatsächlich synthetisiert werden kann. Von der Verwendung des ISE-internen Simulator wird abgeraten, da er beispielsweise die Darstellung von Variablen nicht unterstützt

15 Ergebnisse

Nach klassischer Meinung haben nur Rechner das Potential und die Universalität, um Rechnernetzprotokolle darauf implementieren zu können, da diese komplexe Algorithmen und Datenstrukturen erfordern. Eine Implementierung in Hardware schien nach dieser Meinung ausgeschlossen. Mit dem CarRing II-Projekt wird diese Meinung widerlegt. Dazu war es allerdings erforderlich, eine Methode zu finden, die es erlaubt, VHDL ähnlich wie die prozedurale Sprache C verwenden zu können. Daraus ist der Software-orientierter Programmierstil von VHDL entstanden. Dieser Stil erlaubt ein höheres Abstraktionsniveau als der klassische hardware-orientierte Programmierstil, bei gleichzeitig besserer Testbarkeit und Compilierbarkeit für die spezielle Anwendungsklasse (mathematischer) Algorithmen. Der Stil besteht in der Beschränkung auf einen Subset von VHDL und der Anwendung bestimmter Regeln beim Programmieren. Das Resultat sind Programme, die gut synthetisiert werden können. Simulationen spielen nur als Testhilfe eine Rolle. Mit der algorithmischen Programmierung kann die entstehende Lücke zwischen der FPGA-Hardware einerseits und FPGA-Software andererseits verkleinert werden. Nichtsdestrotz wird für die reine Beschreibung von Hardware der hardware-orientierte Programmierstil seine Bedeutung behalten. Beispielsweise wird die Beschreibung einer CPU auch in Zukunft im konventionellen VHDL-Stil erfolgen. Für Algorithmen und Datenstrukturen ist der Software-orientierte Stil, der im vorliegenden Report in Theorie und Praxis dargestellt wurde, allerdings besser geeignet, was anhand der zahlreichen Code-Beispiele gezeigt werden konnte.

16 Literaturverzeichnis

[1] H. Richter, Elektronik und Datenkommunikation im Automobil, in: IfI Technical Report IfI-09-

05, ISSN: 1860-8477, http://www.in.tu-clausthal.de/de/forschung/technical-reports/, editor: Department of Computer Science, Clausthal University of Technology, Germany, May 2009
[2] M. Wille, H. Richter, C. Asam, Beyond FlexRay - A Survey of CarRing II, Proc. ATACDesign Forum on Automotive Bus Systems (Association for the Advancement of Automotive,) @ International Solid-State Circuits Conference ISSCC 2007, http://www.isscc.org, San Francisco, 11-15. Feb. 2007, Editor: Wolfgang Pribyl, Uni Graz, Publisher: YesEvents, Baltimore, MD, USA, 2007
[3] C. Asam, M. Wille, H. Richter, Carring II: A Real-Time Computer Network as Successor for Flexray?, Proc. Embedded Systems Conference at the Electronica Exhibition, Munich, 14.-15. Nov. 2006
[4] IEEE Computer Society, IEEE Standard VHDL Language, Std 1076-2008, 26 January 2009, New York, IEEE 3 Park Avenue, NY 10016-5997, USA, http://www.ieee.org.
[5] http://de.wikipedia.org/wiki/SystemC
[6] http://en.wikipedia.org/wiki/Handel-C
[7] http://www.mentor.com/products/fpga/handel-c/
[8] Paul Molitor, Jörg Ritter, VHDL Eine Einführung, Pearson-Studium, 2004
[9] Ulrich Heinkel et. al. The VHDL Reference, Wiley, 2002
[10] Peter J. Ashenden, The Designer's Guide to VHDL, 3rd edition, Morgan Kaufmann, 2008
[11] Peter J. Ashenden, VHDL - 2008, Morgan Kaufmann, 2008
[12] Günter Jorke, Rechnergestützter Entwurf digitaler Schaltungen, Fachbuchverlag Leipzig, 2004
[13] Peter Wilson, Design Recipes for FPGAs, Elsevier, 2007
[14] Jürgen Reichardt, Bernd Schwarz, VHDL-Synthese, Oldenbourg (5. Auflage), 2009
[15] James R. Armstrong, F. Gail Gray, Dorling Kindersley, VHDL Design Representation and Synthesis. Pearson, 2007
[16] Andreas Mäder, Vergleichende Untersuchungen zum effizienten Einsatz von VHDL in Simulation und Synthese, Der Andere Verlag, 2004